A Prolog Compiler and its Extension for Or-Parallelism

Mats Carlsson
Swedish Institute of Computer Science
PO Box 1263, S-16428 KISTA, Sweden

Draft version: 19 October 1995

A Prolog Compiler and its Extension for Or-Parallelism

Mats Carlsson
Swedish Institute of Computer Science
P.O. Box 1263, S-16428 KISTA, Sweden

Abstract

This report describes algorithms for the compiler component of the Aurora Or-Parallel Prolog system. The compiler translates one Prolog clause at a time into a sequence of abstract instructions. The instruction set is based on the sequential Warren Abstract Machine (WAM) with extensions for full Prolog, shallow backtracking, memory management and garbage collection, and for the SRI model of or-parallel execution of Prolog.

Most of the described algorithms apply to compilation of sequential Prolog programs. The extensions introduced to support or-parallelism are minor, and concern pruning operators (cut and commit) and compile-time allocation of binding array offsets for permanent variables (generalised environment trimming).

Code generation proper is kept separate from register allocation, and uses heuristics for finding a compilation order which minimises the number of register-register copies. After such copies have been coalesced where possible, register allocation is performed in a single pass over the intermediate code.

The various compilation phases are described in detail, and the implementation is compared with some other compilers.

1. Introduction

This report describes the compiler component of the Aurora Or-Parallel Prolog system [Lusk et al. 88], an implementation based on the SRI model for or-parallel execution [Warren 87]. On the instruction set level, the Aurora abstract machine [Carlsson and Szeredi 90] is much like the sequential Warren Abstract Machine (WAM), [Warren 83]. A synopsis of the WAM appears in Appendix I. In the SRI model, each Prolog process has a private *binding array* as the means for maintaining different bindings for a given variable in different branches of the tree. Allocation of binding array locations for variables is aided by the compiler. This is the only major impact by the SRI model on the instruction set. A synopsis of the Aurora instruction set is given in Appendix II.

The compiler is written entirely in Prolog. It is a *clause compiler*, i.e. it reads one Prolog clause at a time from an input stream, translates that clause to an intermediate representation as a sequence of symbolic instructions for the Aurora abstract machine and passes the intermediate representation to one of several back-ends. The back-ends currently produce bytecode, either in-core or emitted to an object file, to be emulated by the Aurora emulator. Back-ends that produce native code could be produced as well, and this has indeed been done in SICStus Prolog [SICS 88], the sequential Prolog system that Aurora descends from. This report describes the translation from source to intermediate code.

The compiler also computes for each clause certain *indexing information* (see section 3.5, page 25) and certain *static information*, e.g. whether or not the clause contains a cut, or whether the predicate it belongs to has been annotated as sequential. The latter is provided for the benefit of the Aurora scheduler, and is not mentioned further in this report. For details see [Carlsson and Szeredi 90].

The main contributions of this work can be summarised as:

- The WAM *environment trimming* mechanism is generalised in the SRI model to apply to the binding array, by means of new operands which have been added to certain instructions.
- The compiler optimises *shallow backtracking* [Carlsson 89].
- The compiler provides support for *memory management* by detecting situations in which there may be insufficient global stack space, and by ensuring that there can be no undefined pointers in any environment when garbage collection is invoked.
- By using a *heuristic compilation order* when compiling head and body goal arguments, the final code often contains fewer register-register copies than code produced by compiling in strict left-to-right order.

- Code generation proper is done prior to *register allocation*. After coalescing as many register-register copies as possible, the compiler performs register allocation in a single pass over the intermediate code.

The following design decisions are also worth noting:

- When the Aurora emulator calls a predicate P, it performs clause indexing by consulting a stereotypical decision tree, which is updated each time a new clause for P is loaded in core. Consequently, the compiler does not have to emit any indexing instructions, and the overall program structure can be a clause compiler.
- Control structures (disjunctions, if-then-else, and negation by failure) are treated as syntactic sugar for calls to anonymous predicates. The compiler compiles each such control structure just as a user predicate with a special naming convention to avoid clashes.
- The intermediate code is divided into *basic blocks*, i.e. instructions sequences over which temporary registers are live. Register allocation is done one basic block at a time.
- As a consequence of the shallow backtracking optimisation, the compiled code for a clause may have two entrypoints, which merge at some point into a common code stream.

The rest of the report is organised as follows. Chapter 2 introduces our terminology and our extensions of the WAM instruction set. Chapter 3 gives an account of the various translation phases a clause goes through. Chapter 4 describes optimisations involving multiple entrypoints into the compiled code. Chapter 5 contains a comparison with previous work on compiling Prolog and with other register allocation methods. We end with a short concluding chapter.

2. Preliminaries

In this chapter we introduce our terminology, and explain the extensions introduced in the WAM instruction set to support or optimise full Prolog, indexing, memory management, backtracking, and the SRI model.

2.1 Terminology

Each clause is first divided into a number of *chunks*. A chunk is a sequence of *inline goals* followed by a procedure call or by the end of clause. The head counts as part of the first chunk. The following goals are inline goals. They all compile to sequences of special purpose instructions. All other goals, including control structures, are treated as general procedure calls:

```
'$choice'(_)        atom(_)          _ == _           _ =:= _
'$cut'(_)           atomic(_)        _ \== _          _ =\= _
'$commit'(_)        integer(_)       _ @< _           _ < _
_ is _              nonvar(_)        _ @>= _          _ >= _
arg(_, _, _)        number(_)        _ @> _           _ > _
'C'(_, _, _)        var(_)           _ @=< _          _ =< _
compare(_, _, _)    _ = _
functor(_, _, _)    _ =.. _
```

where `'$choice'(_)`, `'$cut'(_)`, and `'$commit'(_)` are *pseudo-goals* corresponding to pruning operators and certain control structures in the source code.

Thus, the clause:

```
p(N, T, Vars, S0, S) :-
        arg(N, T, A),
        p(A, Vars, S0, S1),
        M is N-1,
        p(M, T, Vars, S1, S).
```

is divided into two chunks:

```
p(N, T, Vars, S0, S) :-        % chunk 1
        arg(N, T, A),          % chunk 1
```

```
p(A, Vars, S0, S1),    % chunk 1

M is N-1,              % chunk 2
p(M, T, Vars, S1, S). % chunk 2
```

We statically classify each clause as either *recursive*, if it has more than one chunk, *iterative*, if it has exactly one chunk that ends with a procedure call, or *halting*, if it doesn't contain any procedure calls.

Aurora uses the SRI memory model, in which unbound variables are no longer represented as self-references. Instead, they are represented as references to binding array locations. Thus when a variable is initialised as unbound, a slot in the binding array is reserved for the new variable, enabling different Prolog processes to record different bindings for it.

In the compiler's internal data structures, the *object code name* of a temporary or permanent variable is a term `x(`N`)` or `y(`N`)` respectively, where N is an integer ≥ 0 denoting an offset in the argument register bank or in the current environment, respectively. A special naming convention has been introduced to support pruning operators: the object code name `x(-1)` denotes the value of the current choicepoint register at the most recent procedure call. This somewhat odd object code name is chosen so that the register allocator can perform the same optimisations on this value as on argument registers.

2.2 Instruction Set Extensions

The Aurora instruction set is an extended subset of the WAM instruction set. Notably, no indexing instructions are used; instead, an indexing tree is incrementally built up per predicate as each clause is compiled. At each general procedure call, the emulator dereferences the first input argument (`x(0)`) and dispatches on the indexing tree, instead of executing indexing instructions. In the sections below we shall describe the various extensions. Finally, we introduce the notation `copy(`i`,`j`)` as the canonical representation of the synonymous `get_variable(x(`j`),`i`)` and `put_value(x(`i`),`j`)`.

2.2.1 Support for Full Prolog

The original WAM instruction set does not cover the full language. Efficient implementation of operations such as cut and arithmetic requires extensions so that such operations can be performed in-line. These extensions are provided by the `choice`, `cut`, `function`, and `builtin` instructions.

2.2.2 Indexing Support

Although the WAM indexing instructions are not used, instructions which match the first argument, e.g. `get_constant(`C`,0)`, can sometimes be replaced by specialised versions, e.g. `get_constant_x0(`C`)`, which rely on the fact that the first argument register has been dereferenced and has been found to contain the desired value or an unbound variable.

2.2.3 Shallow Backtracking Support

The key idea of the shallow backtracking optimisation is to postpone or altogether avoid creating choicepoints and restoring states from them. A description of a modified WAM which performs this optimisation appears in [Carlsson 89].

The crucial instruction in the shallow backtracking implementation is the `neck` instruction, which marks the "commitment point" in the clause, i.e. where one must decide whether or not to allocate a choicepoint. There is one `neck` instruction per clause, and it must be placed before any *deep instruction*, where a deep instruction modifies some register whose values must be preserved when backtracking from one clause to another of the same procedure call. In particular, the `neck` instruction must be placed before the `put` sequence of the first procedure call of a non-halting clause, and before any `cut` instruction. Whether or not an instruction is deep sometimes depends on the context, since `put` instructions sometimes alter head arguments and sometimes don't.

The `neck` instruction is actually placed immediately after the last instruction that can cause backtracking (if any), but before any deep instruction. It is pointless to place it after an instruction that cannot fail, and placing it later than necessary may actually degrade the compiler's register allocation.

To enhance the efficiency of shallow backtracking, the WAM `allocate` instruction has been split into `allocate1` and `allocate2`, where `allocate2` is deep and `allocate1` is not. This measure ensures that the value of the current environment register is preserved over the shallow backtracking phase, since that register is left unchanged by `allocate1`.

Finally, the `branch` instruction is introduced to support two entrypoints for some clauses. See chapter 4, page 39 for details.

2.2.4 Memory Management Support

At every procedure call, the emulator checks to see that a certain fixed amount of free memory exists beyond the global stack pointer. To cater for clauses whose first chunk might consume more than this amount, the compiler inserts before all other instructions a `heapmargin_call` instruction to ensure that a sufficient amount exists. Similarly for the first chunk of a clause continuation (always immediately preceded by a `call` instruction), a `heapmargin_exit` instruction is inserted before all other instructions if the chunk might consume more memory than the amount allowed by default.

To support garbage collection, the implementation insists that there be no dangling or undefined pointers in any environment when a general procedure call occurs. This is ensured by initialising all permanent variables in the first chunk and replacing `put_variable`, `get_variable` and `unify_variable` instructions in subsequent chunks by `put_value`, `get_first_value` and `unify_first_value` instructions, respectively. The latter two instructions assume that the permanent variable is currently unbound and ensure by trailing that it will stay unbound if the execution could backtrack to a choicepoint created after the environment. See [Appleby et al. 88] for details about the garbage collection algorithm.

2.2.5 SRI Model Support

In the Aurora abstract machine, environment trimming is generalised to apply to the binding array as well as to the local stack. Each environment contains a base pointer to a *shadow environment* in the binding array. Only those permanent variables that are initialised as local, i.e. whose first occurrence compiles to a `put_variable` or a `put_void` instruction (see below), need slots in the shadow environment. The size of the portion of the shadow environment that must be kept after a call is called the *active variable count* and occurs as the *VarCount* operand below. The size of the portion of the actual environment that must be kept is called the *active environment size* and occurs as *EnvSize* below.

Local unbound variables are created by a combination of run-time and compile-time allocation. The `put_variable` and `put_void` instructions acquire for permanent variables a *variable number* operand (the *VarNo* operand below), which denotes the variable's offset in the shadow environment. The binding array location of a global unbound variable is determined at run time. A new machine register has been introduced for this purpose, and maintains the next available binding array location.

The full format of the augmented instructions is thus:

`call(`*Pred*`,`*EnvSize*`,`*VarCount*`)`

`put_variable(`*Vn*`,`*Ai*`,`*VarNo*`)`
: where *VarNo* is ignored for temporary variables.

`put_void(`*Vn*`,`*VarNo*`)`
: where this instruction replaces `put_variable` if *Ai* has no use.

There are two *pruning operators* currently supported by Aurora: the conventional Prolog *cut*, which prunes all branches to its right, and a symmetric version of cut called *commit*, which prunes branches both to its left and right. Pruning operators compile to `cut` instructions, which acquire an extra argument to distinguish the type of pruning operation (cut or commit). Pruning operators can cause the current computation to suspend in Aurora, in which case the current state must be saved into a data structure. Since all live temporary variables must be saved as part of the current state, `cut` instructions also need an extra argument denoting which temporary variables are currently live.

3. Compilation Phases

Compilation of a clause proceeds through a number of phases. In this chapter we give an account of those phases. The following processing phases can be identified.

- The source code for a clause is converted to an abstract syntax tree, which is a more suitable representation for the rest of the compiler. Certain static information, e.g. regarding the presence of cuts in a clause, is gathered from the abstract syntax tree.
- Certain control structures, such as disjunctions, negations, and if-then-else constructs, are broken off as internal predicates, to be recursively compiled.
- Variables are classified as temporary or permanent, and permanent variables are allocated.
- The abstract syntax tree is translated to a sequence of instructions, and is not needed any more in the processing of the current clause.
- Indexing information is extracted from the generated code, and certain transformations are performed in order to aid register allocation.
- Register allocation is performed for the temporary variables of the clause. This allocation is performed per chunk, and involves computing live variable sets for each instruction.
- A final editing phase applies certain local changes and optimisations to the code.

For some back-ends there would be an extra phase when all clauses have been processed. This phase would emit indexing instructions for all predicates encountered, for example when compiling to native code. Such a phase has been implemented for the native code back-end of SICStus Prolog, but is not discussed in this report. Indexing methods for the WAM are discussed in [Carlsson 87].

3.1 Generating the Abstract Syntax Tree

In order to make the source code more tractable to the compiler, in particular variable occurrences, the source code for a clause of a predicate P is converted to an *abstract syntax tree*.

This phase specially recognises cuts, and transforms them to pseudo-goals `'$cut'(`B`)`. Commits are similarly transformed to pseudo-goals `'$commit'(`B`)`. A pseudo-goal `'$choice'(`B`)` is also added as the first goal of the clause. A later phase will compile the latter pseudo-goal to code that loads B with a value representing the current choicepoint when P was called, i.e. the youngest choicepoint that the cut must preserve. For example, a clause

```
p(X, Y) :-
        integer(X), !,
        q(X, Y).
```

is transformed to

```
p(X, Y) :-
        '$choice'(C),
        integer(X),
        '$cut'(C),
        q(X, Y).
```

Certain control structures also expand to abstract syntax trees involving these pseudo-goals.

The abstract syntax trees are represented as:

`var(`*D*`,`*N*`)`
: This representation denotes a source variable *U*. *D* is used as a *variable descriptor*, and is eventually instantiated by the compiler to one of the values:

  `g` (for global), if the first occurrence of *U* compiles to a `unify_variable` instruction or, if *U* is a temporary variable, to a `put_variable` instruction. In either case, the variable is guaranteed not to hold a reference to the local stack at run time.

  `r` (for remote), if the first occurrence of *U* compiles to a `get_variable` instruction. In this case, the variable is guaranteed not to hold a reference to the current environment at run time.

  I where *I* is an integer, if the first occurrence of *U* compiles to a `put_variable` instruction and *U* is a permanent variable whose object code name is `y(`*I*`)`. In this case, the variable will initially be unbound, and the compiler can use *I* to ensure that older local variables are never bound to newer local variables.

  N is eventually instantiated to the *U*'s object code name. It is left uninstantiated by this phase.

`constant(`*K*`)`
: This representation denotes the atomic source term *K* except the empty list.

`nil`
: This representation denotes the empty list.

`list(`*X*`,`*Y*`)`
: This representation denotes the source list `[`*X0*`|`*Y0*`]`, where *X* and *Y* are abstract syntax trees denoting the arguments *X0* and *Y0*.

`structure(`*F*`,`*L*`)`
: This representation denotes a source structure *F(A1,...,An)*, where *L* is a list of abstract syntax trees denoting the arguments *A1,...,An* of the structure.

For example, the `p/2` clause above is represented as the abstract syntax tree:

```
% head
structure(p,[var(DX,X),var(DY,Y)]).

% body
structure(',',
          [structure('$choice',[var(DB,B)]),
           structure(',',
                     [structure(integer,[var(DX,X)]),
                      structure(',',
                                [structure('$cut',[var(DB,B)]),
                                 structure(q,[var(DX,X),var(DY,Y)])])])]).
```

3.2 Spawning Internal Predicates

Here, certain control structures are recognised and are broken out as internal predicates. Each occurrence is replaced by a call to such an internal predicate. Conjunctions are linearised, so that the resulting body data structure becomes a list of goals rather than a tree. For instance, the `p/2` clause of the previous section becomes:

```
% head
structure(p,[var(DX,X),var(DY,Y)]).

% body
[structure('$choice',[var(DB,B)]),
 structure(integer,[var(DX,X)]),
 structure('$cut',[var(DB,B)]),
 structure(q,[var(DX,X),var(DY,Y)])].
```

This phase also processes type tests of the first head argument and cuts that occur immediately after the head. Under certain conditions, these can be transformed into indexing information and do not appear in the clause code. This is discussed in detail in see section 3.5, page 25.

The control structures that generate internal predicates are:

`(`*P* `;` *Q*`)` denoting disjunctions,

`(`*If* `->` *Then*`)`
`(`*If* `->` *Then* `;` *Else*`)`
denoting if-then-else constructs,

`(\+` *Goal*`)` denoting negation as failure.

Whenever one of the above control structures is encountered, it is transformed to a call to an internal predicate $P(\Omega)$, where Ω is the set of variables that are shared between the control structure and the rest of the clause. The transformation procedure produces a list of clauses for P, and can be described by the following schematic DCG grammar:

```
disj((P ; Q), Ω) -->
        subdisj(P, Ω), disj(Q, Ω).
disj((If -> Then), Ω) -->
        [(P(Ω) :- '$choice'(B), If, '$cut'(B), Then)].
disj((\+ Goal), Ω) -->
        disj((Goal -> fail ; true), Ω).
disj(Goal, Ω) -->
        { Goal is not of any of the above types },
        [(P(Ω) :- Goal)].


subdisj((If -> Then), Ω) -->
        [(P(Ω) :- '$choice'(B), If, '$cut'(B), Then)].
subdisj(Goal, Ω) -->
        { Goal is not of the above type },
        [(P(Ω) :- Goal)].
```

Thus the transformation rules for if-then-else and negation as failure constructs involve cuts which are local to the internal predicate P: each cut is linked by the B argument to a matching `$choice` pseudo-goal, which defines the scope of the cut. The scope of a cut occurring in the source code extends beyond all internal predicates, and must not appear in any of the *If* parts above. If cuts were allowed in an *If* part, their scope would not be properly included in the scope of the local cut.

The two disjuncts of `(`P `;` Q`)` must be treated differently, since ';' is not associative. For example, the two expressions

```
G1 :-
        ((P -> Q ; R) ; S).
```

```
G2 :-
        (P -> Q ; (R ; S)).
```

---

are not equivalent: in the former case the scope of the local cut does not include `S`; in the latter case it does. The result after expansion is:

```
G1 :- P0001.

P0001 :- P0002.
P0001 :- S.

P0002 :- '$choice'(B), P, '$cut'(B), Q.
P0002 :- R.

G2 :- P0003.

P0003 :- '$choice'(B), P, '$cut'(B), Q.
P0003 :- R.
P0003 :- S.
```

The compiler is invoked recursively on each spawned internal predicate. The advantage of this strategy of compiling control structures is its simplicity: there are no branches in the code for a clause, which greatly simplifies lifetime analysis. Updating variable descriptors becomes straightforward too. The disadvantage is that some opportunities for optimisation are sacrificed.

This is in contrast to the approach taken e.g. in the Berkeley PLM compiler [Van Roy 84] where disjunctions are coded in-line. Van Roy avoids sometimes having to allocate extra environments, but the compile-time analyses become complicated by the code being non-linear, and the implementation of cut runs into some extra trouble.

In general, transformations such as the above are perhaps better handled by a precompiler. This approach has been taken in the RAP-WAM compiler [Hermenegildo 88], which was produced by taking the SICStus compiler and extending it to generate the *conditional graph expressions* needed for supporting restricted and-parallelism. A precompiler could also do more extensive transformations such as partial evaluation in a more comprehensive manner.

3.3 Allocating Permanent Variables

Here, variables are classified as temporary or permanent. Object code names are assigned to the permanent variables.

Any variable that occurs in more than one chunk is permanent. Permanent variables are arranged such that they can be discarded as soon as possible. This is achieved by treating each goal separately in reverse order, from the last to the first. Each goal is traversed depth first, from left to right, and each time a permanent variable is encountered that has not yet been allocated, it is assigned a number. The numbers are assigned in increasing order. See Appendix I for an example.

3.4 Code Generation Proper

Each clause is translated to a sequence of instructions in a fairly naive way. No `allocate1`, `allocate2`, or `deallocate` instructions are emitted here; instead, these instructions are filled in by the final editing phase, and so is the *VarCount* field required by the SRI model in the `put_variable(y(_),_,_)`, `put_void(y(_),_)`, and `call(_,_,_)` instructions.

This phase leaves temporary variables unallocated (represented as unbound variables). Eventually, a register number gets assigned to each temporary variable (see section 3.6, page 28). Singleton variables are not distinguished from temporary ones. The general strategy is to generate worst-case code, in terms of the number of instructions, hoping that the register allocator is able to turn many instructions into no-ops and excise them.

The temporary variables of a Prolog clause and new temporary variables introduced by flattening compound terms and by compiling certain inline goals are collectively called *temporaries*. The temporaries introduced by the code generation phase have an important property: *each temporary denotes a unique value*. This property is called the *single assignment* property and will be used in the register allocation phase. It follows from two facts:

- Temporaries are not reused, i.e. each time a temporary is introduced, it is distinct from all previously used temporaries. In implementation terms, it is a unique logical variable.
- The instruction set has been designed so that once a temporary variable has acquired a value, no instruction replaces that value by another value. For instance, the `function(`*F*`,`*T3*`,`*T1*`,`*T2*`)` instruction explicitly specifies the destination: it computes *T3* as the value of applying *F* to the arguments *T1* and *T2*, as opposed to storing the value into *T1* or *T2*.

The single assignment property does not generally apply to argument registers, since they are used for both head and goal arguments. Depending on the context, however, the property may apply to argument registers:

- In the first chunk of a clause, argument registers which are not used both in the head and in the procedure call denote unique values.
- In subsequent chunks, argument registers are only used for procedure calls and do denote unique values.

3.4.1 Overview

The general picture of clause compilation is best described by an example. Consider a clause with three chunks:

```
Head :-
        InlineGoal1,
        p(...),
        InlineGoal2,
        q(...),
        InlineGoal3,
        r(...).
```

which compiles to code according to the pattern:

```
heapmargin_call(H1,A)
get sequence for Head
code for InlineGoal1
neck(A)
put sequence for p
call(p/N1,K1,M1)
heapmargin_exit(H2)
code for InlineGoal2
put sequence for q
call(q/N2,K2,M2)
heapmargin_exit(H3)
code for InlineGoal3
put sequence for r
execute(r/N3)
```

where *A* is the arity of the clause; *H1*, *H2*, and *H3* are the maximum amounts of global stack space consumed by the chunks 1, 2, and 3, respectively; *N1*, *N2*, and *N3* are the arities of `p`, `q`, and `r`, respectively; *K1* and *K2* are the active environment sizes after the respective calls; and *M1* and *M2* are the active variable counts after the respective calls. Thus all chunks are terminated by a `call` instruction except the last one, which is terminated by an `execute` instruction. Halting clauses are treated as having a single body goal `true`.

All chunks begin with a `heapmargin_exit` instruction except the first one, which begins with a `heapmargin_call` instruction. The final editing phase transforms `execute(true/0)` into `proceed` and deletes `heapmargin`... instructions if the arguments are below certain thresholds.

The `neck` instruction is initially placed immediately preceding the put sequence for the first procedure call of the clause. A later phase will move the `neck` to its appropriate position (see section 3.5, page 25).

The main work of this phase consists in emitting code for creating the arguments of body goals and for matching goal arguments against head arguments. There is considerable freedom in choosing which order head and goal arguments should be compiled in. Compiling head and goal arguments in textual order often yields suboptimal code. The objective is of course to generate as good code as possible at reasonable cost. As it is hardly realistic to achieve truly optimal code generation, we have resorted to heuristics which give considerable improvements over compiling in textual order. For example,

```
p([X|Y],Y) :- q(Z,[X|Z]).
```

compiles to:

```
(1)     heapmargin_call(4,2)    % p(A,B) :-
(2)     get_variable(x(Y),1)    %  B=Y,
(3)     get_list(0)             %  A=[
(4)     unify_variable(x(X))    %     X|
(5)     unify_local_value(x(1))%       B]
(6)     neck(2)                 %  ,
(7)     put_list(1)             %  B=[
(8)     unify_value(x(X))       %      X|
(9)     unify_variable(x(Z))    %        Z],
(10)    put_value(x(Z),0)       %  A=Z,
(11)    execute(q/2)            %  q(A,B).
```

where in a later pass X, Y and Z are assigned register numbers 2, 1, and 0, respectively, and so instructions (1), (2) and (10) can be deleted, as the `heapmargin` argument is well below the threshold. Notice that instruction (5) uses input argument register 1 (`B`) instead of the temporary Y introduced in instruction (2). This is due to maintaining a compile-time cache of argument registers and their values (see below).

Another way of describing the desired effect of the heuristic order of the code for head and goal unification is to minimise overlapping lifetimes of temporary variables and argument registers. See section 3.6.1, page 28.

During code generation, knowledge about argument register contents is maintained and used as a compile-time *argument register cache*. The cache is valid throughout a chunk. There are several reasons for maintaining the cache:

- It is usually cheaper to access argument registers than permanent variables or constants, particularly in a native code implementation, and so strength reduction can be applied to certain operations.
- The argument register cache can sometimes be used to determine that a goal argument register already contains the desired term, in which case no code is emitted.
- For permanent variables occurring as head arguments, accesses in the first chunk to such variables (`get_value(y(_),_)`, `unify_value(y(_))`, `put_value(y(_),_)`) can often be replaced by accesses to the respective argument register (`get_value(x(_),_)`, `unify_value(x(_))`, `put_value(x(_),_)`). This makes it possible to postpone the `get_variable(y(_),_)` instruction that initialises the permanent variable, and sometimes the `allocate1` instruction, until a point in the code where it is known that the head unification has succeeded.
- One of the code transformations of a later phase (see section 3.5, page 25) requires that temporary variables *Ai* initialised by `get_variable(`*Ai*`,`*B*`)` must not be used before the `neck` instruction. The cache mechanism guarantees that all such uses of *Ai* are replaced by uses of *B*.

The following sections will treat code generation for head arguments, body arguments, structure arguments and inline goals. In the descriptions, the notation `cache[`*i*`]` will be used to denote the cached value for argument register *i*. The notation `void` is used to denote an invalid cache value. When a chunk is entered, all cache values are initially `void`.

3.4.2 Flattening Terms

In the original WAM, compound terms had to be fully *flattened* by introducing new temporary variables. For example, a clause

```
p(f(X,g(Y),h(Z))) :- p(f(Y,g(Z),h(X))).
```

had to be compiled as

```
p(f(X,X1,X2)) :-
        X1 = g(Y),
        X2 = h(Z),
        Y1 = g(Z),
        Y2 = h(X),
        p(f(Y,Y1,Y2)).
```

In our extended WAM we introduced the `unify_list` and `unify_structure(`S`)` instructions, to be used if the *last* subterm of a compound term is itself a compound term. Thus the above clause could be compiled as:

```
p(f(X,X1,h(Z))) :-
        X1 = g(Y),
        Y1 = g(Z),
        p(f(Y,Y1,h(X))).
```

However, use of `unify_list` and `unify_structure(`S`)` is restricted to compound terms in which *only* the last subterm is a compound term. Otherwise it is easy to construct non-contrived cases that need an unbounded number of temporary variables. For example, with an unrestricted use of these instructions, the clause:

```
p([a1-1,a2-2,a3-3,...,aN-N]).
```

would be compiled as

```
p([T1,T2,T3,...,TN]) :-
        T1 = a1-1,
        T2 = a2-2,
        T3 = a3-3, ...,
        TN = aN-N.
```

which requires N argument registers. With a restricted use, it would be compiled as:

```
p([H1|T1]) :-
        H1 = a1-1,
        T1 = [H2|T2],
```

```
H2 = a2-2,
T2 = [H3|T3],
H3 = a3-3, ...,
TN-1 = [HN],
HN = aN-N.
```

which although it requires $2N$ temporaries only requires 2 actual registers, as all Hi can be assigned register `x(0)` and all Ti can be assigned register `x(1)`. This observation is due to Takashi Chikayama.

3.4.3 Head Arguments

The head arguments are compiled using the simple heuristic that arguments which are variables are compiled *before* other head arguments, and from right to left. This unification order increases the set of available registers when compound terms are considered, and also makes it possible to condense sequences like:

```
get_list(0)                 % p([
unify_variable(x(M))        %    X
unify_nil                   %     ],
get_value(x(M),1)           %       X) :- ...
```

into

```
get_variable(x(M),1)        % p(    X,
get_list(0)                 %   [
unify_local_value(x(M))     %    X
unify_nil                   %     ]  ) :- ...
```

where in the latter sequence the `get_variable` instruction becomes a no-op and is deleted, if `M` can be assigned the register number 1.

For each argument position i in the heuristic order, argument number i is first flattened, as described above. The code generation then follows a case analysis on the abstract syntactic structure of that argument:

`var(`D`,`N`)`

```
if (first occurrence of N)
  D = r,
  emit get_variable(N,i),
  let cache[i] = var(r,N)
else if (∃ j : cache[j] ≡ var(D',N))
  emit get_value(x(j),i),
  let cache[i] = var(D',N)
else
  emit get_value(N,i),
  let cache[i] = var(D,N)
```

`constant(`K`)`

```
  emit get_constant(K,i),
  let cache[i] = constant(K)
```

`nil`

```
  emit get_nil(i),
  let cache[i] = nil
```

`list(`X`,`Y`)`

```
  emit get_list(i),
  emit code for X and Y
```

`structure(`F`,`L`)`

```
  emit get_structure(F/A,i),
  emit code for all elements of L
```

---

where A is the length of L.

3.4.4 Body Goal Arguments

The goal arguments are compiled using the simple heuristic that arguments which are variables are compiled *after* other goal arguments, and from left to right. As for head arguments, this unification order increases the set of available registers when compound terms are considered, and also makes it possible to condense sequences like:

```
put_variable(x(M),0,_)          % :- p(X,
put_list(1)                     %       [
unify_value(x(M))               %        X
unify_nil                       %         ]), ...
```

into

```
put_list(1)                     % :- p(  [
unify_variable(x(M))            %        X
unify_nil                       %         ],
put_value(x(M),0,_)             %     X    ), ...
```

where in the latter sequence the `put_value` instruction becomes a no-op and is deleted, if `M` can be assigned the register number 0, like in the head compilation case.

For each argument position i in the heuristic order, that argument is first flattened, as described above. The code generation then follows a case analysis on the abstract syntactic structure of that argument, where k is the active environment size after the body goal that terminates the current chunk:

`var(`D`,`N`)`

```
if (first occurrence of N)
  if (N ≡ y(I)
    D = I
  else
    D = g,
  emit put_variable(N,i,_),
  let cache[i] = var(D,N)
else if (∃ j : cache[j] ≡ var(D',N) ∧ (D' ≡ g ∨ D' ≡ r ∨ D'<k))
  emit put_value(x(j),i),
  let cache[i] = var(D',N)
else if (∃ j : cache[j] ≡ var(D',N))
  emit put_unsafe_value(x(j),i),
  let cache[i] = var(r,N)
else if (D ≡ g ∨ D ≡ r ∨ D<k)
  emit put_value(N,i),
  let cache[i] = var(D,N)
else
  emit put_unsafe_value(N,i),
  let cache[i] = var(r,N)
```

`constant(`K`)`

```
if (∃ j : cache[j] ≡ constant(K))
  emit put_value(x(j),i),
  let cache[i] = constant(K)
else
  emit put_constant(K,i),
  let cache[i] = constant(K)
```

`nil`

```
if (∃ j : cache[j] ≡ nil)
  emit put_value(x(j),i),
  let cache[i] = nil
else
  emit put_nil(i),
  let cache[i] = nil
```

`list(`X`,`Y`)`

```
  emit put_list(i),
  let cache[i] = void,
  emit code for X and Y
```

`structure(`F`,`L`)`

```
  emit put_structure(F/A,i),
  let cache[i] = void,
  emit code for all elements of L
```

where A is the length of L.

3.4.5 Compound Terms

For each structure argument, the code generation follows a case analysis on the abstract syntactic structure of that argument:

`var(`D`,`N`)`

```
if (first occurrence of N)
  D = g,
  emit unify_variable(N)
else if (exists j : cache[j] ≡ var(g,N))
  emit unify_value(x(j))
else if (exists j : cache[j] ≡ var(D',N))
  emit unify_local_value(x(j))
```

```
else if (D ≡ g)
  emit unify_value(N)
else
  emit unify_local_value(N)
```

`constant(`K`)`

```
  emit unify_constant(K)
```

`nil`

```
  emit unify_nil,
```

`list(`X`,`Y`)`

```
  emit unify_list,
  emit code for X and Y
```

`structure(`F`,`L`)`

```
  emit unify_structure(F/A),
  emit code for all elements of L
```

---

where A is the length of L.

3.4.6 Inline Goals

For many inline goals, the code generated is rather like the code for an ordinary procedure call: the arguments are set up, except the argument registers are not constrained to particular numbers, and a utility instruction for the actual goal is emitted. Some inline goals, however, have their own nonstandard compilation methods:

`'$cut'(var(_,`N`))`
`'$commit'(var(_,`N`))`
: The code emitted is `cut(`N`,`L`,`Op`)`, where L is eventually instantiated to the number of currently live argument registers, and Op distinguishes cuts from commits. If the cut occurs in the first chunk of a clause, the final editing phase will later reduce the strength of this instruction to a `cut(`L`,`Op`)` instruction (see section 3.7, page 36).

`'$choice'(var(_,`*N*`))`

The code emitted is `get_variable(`*N*`,-1)`. If there are no pruning operators after the first chunk, the `get` instruction becomes redundant and the final editing phase will excise it. Otherwise, the final editing phase will convert the instruction to a `choice(`*N*`)` instruction.

X `=` *Y*

The code generation can be expressed as a decision table, based on the abstract syntax of the two arguments. Let `VARIABLE` denote the first occurrence of a variable, `VALUE` a non-first variable occurrence, `CONSTANT` an atomic term, and `COMPOUND` a compound term. The table is defined as:

where the actions are described below. For clarity, details about the argument register cache are not included in this account:

1. The compiler takes pains to avoid if possible creating a global variable, and must not create an illegal binding in the local stack.

   If *X* = `y(I)` and *Y* = `y(J)` and *I* `<` *J*, the emitted code is `put_variable(`*X*`,`*i*`,_)` followed by `get_variable(`*Y*`,`*i*`)`.

   Otherwise, the emitted code is `put_variable(`*Y*`,`*i*`,_)` followed by `get_variable(`*X*`,`*i*`)`.

2. The compiler takes pains to avoid if possible creating a local variable, and must not create an illegal binding in the local stack.

   If *VARIABLE* is `x(`*i*`)`, the emitted code is simply `put_value(`*VALUE*`,`*i*`)`.

   Otherwise, if *VALUE* cannot point to the local stack, the emitted code is simply `put_value(`*VALUE*`,`*i*`)` followed by `get_variable(`*VARIABLE*`,`*i*`)`.

   Otherwise, the emitted code is `put_variable(`*VARIABLE*`,`*i*`,_)` followed by `get_value(`*VALUE*`,`*i*`)`.

3. The emitted code is `put_variable(`*VARIABLE*`,`*i*`,_)` followed by code as for compiling head argument *i*.

4. The emitted code is `put_value(`*VALUE*`,`*i*`)` followed by code as for compiling head argument *i*.

5. If the principal functors of the two terms do not match, a `fail` instruction is emitted, otherwise the compiler treats the terms recursively if they are compound terms.
6. A `fail` instruction is emitted.

`'C'(`*A*`, `*B*`, `*C*`)`
: Compiled as *A* `= [`*B*`|`*C*`]`.

Value `is` *Expr*
: Compiled as code for evaluating the arithmetic expression *Expr* into a temporary *Temp* followed for code for a goal *Value* `=` *Temp*. If *Expr* is a variable, code is emitted for evaluating the arithmetic expression `+`*Expr* into *Temp* instead.

  Compilation of an expression *Expr* into a temporary *Temp* proceeds as follows, discriminating on the abstract syntax of *Temp*:

  `structure(`*F*`,[`*X*`])`
  : where *F* is a unary arithmetic operator: *X* is recursively compiled into a temporary *T* and a `function(`*F*`,`*Temp*`,`*T*`)` instruction is emitted.

  `structure(`*F*`,[`*X*`,`*Y*`])`
  : where *F* is a binary arithmetic operator: *X* and *Y* are recursively compiled into temporaries *T* and *U* and a `function(`*F*`,`*Temp*`,`*T*`,`*U*`)` instruction is emitted.

  `list(`*X*`,`*Y*`)`
  : is treated as compiling the expression *X* into *Temp*.

  `constant(`*K*`)`
  : where *K* is a number: compiled to a `put_constant(`*K*`,`*Temp*`)` instruction.

  `var(`*D*`,`*N*`)`
  : compiled to a `put_value(`*N*`,`*Temp*`)` instruction.

`arg(`*A*`, `*B*`, `*C*`)`
`compare(`*C*`, `*A*`, `*B*`)`
: Compiled as code for putting two body goal arguments *A* and *B* into temporaries *T1* and *T2*, followed by a `function(`*F*`,`*T3*`,`*T1*`,`*T2*`)` instruction, where *F* is the predicate name, followed by code for a goal *C* `=` *T3*.

`atom(`*A*`)`
`atomic(`*A*`)`
`integer(`*A*`)`
`nonvar(`*A*`)`
`number(`*A*`)`
`var(`*A*`)`
: Compiled as code for putting a body goal argument *A* into a temporary *T1*, followed by a `builtin(`*F*`,`*T1*`)` instruction, where *F* is the predicate name.

A `==` *B*
A `\==` *B*
A `@<` *B*
A `@>=` *B*

A `@>` *B*

A `@=<` *B*

A `=..` *B* Compiled as code for putting two body goal argument *A* and *B* into temporaries *T1* and *T2*, followed by a `builtin(`*F*`,`*T1*`,`*T2*`)` instruction, where *F* is the predicate name.

A `=:=` *B*

A `=\=` *B*

A `<` *B*

A `>=` *B*

A `>` *B*

A `=<` *B* Compiled as code for evaluating two arithmetic expressions *A* and *B* into temporaries *T1* and *T2*, followed by a `builtin(`*F*`,`*T1*`,`*T2*`)` instruction, where *F* is the predicate name.

`functor(`*A*`,`*B*`,`*C*`)`
Compiled as code for putting three body goal argument *A*, *B*, and *C* into temporaries *T1*, *T2*, and *T3*, followed by a `builtin(functor,`*T1*`,`*T2*`,`*T3*`)` instruction.

3.5 Extracting Indexing Information

The clause indexing mechanism built into the Aurora emulator requires information about which values the first head argument can take in order to match a given clause. The task of this phase is to extract such information, and also to perform certain transformations on the generated code.

The indexing information is extracted in part from the abstract syntax tree and in part from the generated code. The extracted information consists of two components:

A type mask
This encodes which type the first argument may have if this clause is a candidate for a match. The type mask is represented as an integer whose bits encode the types *variable*, *number*, *atom*, *list*, and *structure*.

A key specifier
This encodes which principal functor an instantiated first argument may have if this clause is a candidate for a match. The key specifier is represented as:

`nohash` denoting no constraint on the principal functor,

`hash(`*K*`)` denoting the constraint that the principal functor must be *K*.

The above information is used solely for indexing purposes, and is not present in the final clause code. The type mask is partly extracted from the abstract syntax tree, as part of the process of breaking off control structures (see section 3.2, page 10). If the first head argument is written as a variable *X*, the following situations are recognised in the abstract syntax tree:

- Simple type tests on X that occur immediately after the head by the predicates `var/1`, `nonvar/1`, `atom/1`, `atomic/1`, or `number/1`, or combinations composed by `\+` *T1*, `(`*T1*`,`*T2*`)`, or `(`*T1*`;`*T2*`)`, where *T1* and *T2* are simple or composed type tests, are recognised.

  If all head arguments are unique variables, such type tests disappear from the clause code and become encoded in the type mask. If not all head arguments are unique variables, the head unification could bind X due to sharing between variables, and so the type mask mechanism has to allow unbound variables followed by an explicit test. Thus, type tests are encoded in the type mask as well as in the clause code, but the *variable* bit is always set in the type mask.
- If all head arguments are unique variables, a cut immediately after the head, possibly preceded by type tests on X, is specially optimised: Instead of emitting a cut instruction in the clause code, the information about the cut is stored in the type mask as an extra bit, and the clause indexing mechanism is responsible for "executing" an implicit cut before entering the clause code.

The key specifier is extracted from the generated WAM code (see section 3.4, page 13), which may contribute to the type mask too. If the first WAM instruction I is a *special get instruction*, possibly preceded by `get_variable` instructions, and no implicit cut is in effect, I is replaced by an *indexed get instruction* according to the table:

Special get instruction	*Indexed get instruction*	*Key specifier*
`get_constant(K,0)`	`get_constant_x0(K)`	`hash(K)`
`get_nil(0)`	`get_nil_x0`	`hash([])`
`get_structure(F,0)`	`get_structure_x0(F)`	`hash(F)`
`get_list(0)`	`get_list_x0`	`hash('.'/2)`

Executing an indexed get instruction is considerably cheaper than executing an ordinary get instruction. Indexed get instructions know that `x(0)` has already been dereferenced and either contain the correct value or contain an unbound variable. Consider e.g. the second clause of `concatenate/3` from Appendix I:

```
concatenate([X|L1], L2, [X|L3]) :-
        concatenate(L1, L2, L3).
```

which previous phases have compiled to:

```
    ( 1)get_variable(x(L2),1)   % concatenate(       L2,
    ( 2)get_list(0)             %              [
    ( 3)unify_variable(x(X))    %               X|
```

```
( 4)unify_variable(x(L1))   %                L1],
( 5)get_list(2)             %                    [
( 6)unify_value(x(X))       %                     X|
( 7)unify_variable(x(L3))   %                       L3])
( 8)neck(3)                 % :-
( 9)put_value(x(L1),0)      %     concatenate(L1,
(10)put_value(x(1),1)       %                 L2,
(11)put_value(x(L3),2)      %                    L3)
(12)execute(concatenate/3)  % .
```

---

where for simplicity we have named each temporary variable by its source code name. Notice that in instruction (10), the argument register cache has replaced a use of `x(L2)` by a use of `x(1)`, since instruction (1) copies the latter into the former. This phase recognises the `get_list(0)` instruction and replaces it by `get_list_x0`, producing the type mask <*list,variable*> and key specifier `hash('.'/2)`.

Although not strictly relevant to extracting indexing information, other transformations on the first chunk are also done in this phase:

- If any `cut` instruction occurs before the `neck`, the `neck` is placed immediately before the `cut`. This transformation is repeated several times if necessary.
- If the `neck` is immediately preceded by an instruction that cannot fail, the two instructions are transposed. This transformation is repeated several times if necessary.
- All `get_variable(`A`,`B`)` instructions that correspond to head arguments that are variables or to `$choice` pseudo-goals are moved to after the `neck`. In these instructions B is always instantiated to an argument register number, and A either represents a permanent variable or a temporary variable which is not yet instantiated to a register number. It is easy to see that this transformation preserves semantics by considering the effect of moving one such instruction:
  - A is not used by any instruction up to the `neck`, since all uses of A become replaced by uses of B at least up to the `put` sequence of the first procedure call. This is ensured by the argument register cache.
  - A is not used by the `neck` itself, as A cannot denote an input argument register.
  - No instruction up to the `neck` defines B. This follows from the fact that all input arguments are valid at least up to the `neck`.

  Thus each such `get` instruction can be deferred to after the `neck` instruction. This transformation helps avoid doing useless work in case the head unification fails, and makes it possible for the Aurora emulator to arrange the compiled code so that the indexed get instruction (if any) is skipped if `x(0)` is found to contain the correct value (see chapter 4, page 39).

As explained earlier (see section 2.2.3, page 5), the `neck` instruction must be placed before the `put` sequence for the first procedure call and before any `cut` instructions.

The net effect of the above transformations on the `concatenate/3` clause is thus:

```
get_list_x0
unify_variable(x(X))
unify_variable(x(L1))
get_list(2)
unify_value(x(X))
neck(3)
get_variable(x(L2),1)
unify_variable(x(L3))
put_value(x(L1),0)
put_value(x(1),1)
put_value(x(L3),2)
execute(concatenate/3)
```

3.6 Register Allocation

The task of register allocation is to find a *correct* and *optimal* assignment of register numbers to temporary variables, where an assignment is a function A, mapping temporary variables to register numbers and register numbers to themselves. As a first approximation to the notion of a correct assignment A we require that for all program points where two temporaries or registers have a next use, A must not map them to the same register. We shall relax this requirement later. To meet the requirement, we must compute for each program point the set of temporary variables and argument registers that are *live* at that point, i.e. that hold a value which is used by that or a subsequent instruction. These sets are called *live variable sets*. Using this information we can often collapse many `copy` instructions by assigning the same register to both operands. Finally remaining temporaries are assigned register numbers.

3.6.1 Lifetime Analysis

For each program point, we compute the set of live variables at that point. We use the standard technique (see e.g. [Aho et al. 85]) of walking the code in reverse order, adding or deleting such temporaries and registers from the current set that are used or defined by each instruction, respectively.

We first form a sequence of *use-def operators* $\mathrm{ud}(0)\ldots\mathrm{ud}(m)$ as an abstraction of the instruction sequence, by translating each instruction according to the table in Appendix III. We use the operators:

`d(`i`)` which denotes an occurrence of a temporary or register which is defined by some instruction,

`u(`i`)` which denotes a use of a temporary or register, and

`c(`i`,`j`)` which denotes a `copy(`i`,`j`)` instruction.

where copies are treated specially since a copy operation is useless if the target register has no next use.

Using the *ud* sequence in reverse order, we then form a sequence of live variable sets $\mathrm{lv}(0) \ldots \mathrm{lv}(m{+}1)$ as follows:

$$\begin{aligned} \mathrm{lv}(m{+}1) &= \emptyset, \\ \mathrm{lv}(k) &= \mathrm{f}(\mathrm{lv}(k{+}1), \mathrm{ud}(k)), 0 \leq k \leq m. \end{aligned}$$

where the function f is defined as:

$$\begin{aligned} \mathrm{f}(S, \mathtt{d}(i)) &= S \ominus i, \\ \mathrm{f}(S, \mathtt{u}(i)) &= S \oplus i, \\ \mathrm{f}(S, \mathtt{c}(i,j)) &= S \ominus j \oplus i, j \in S, \\ \mathrm{f}(S, \mathtt{c}(i,j)) &= S, j \notin S. \end{aligned}$$

where $S \ominus i$ and $S \oplus i$ denote set deletion and insertion, respectively.

The first rule for `c(`i`,`j`)` corresponds to a useful copy: j is going to be used after the copy, and so i must be added to the set of registers which are live when the copy is executed. The second rule for `c(`i`,`j`)` corresponds to a useless copy: j is not going to be used afterwards, so there is no point in executing the copy nor in adding i to the live variable set.

Procedure calls are treated as *defining* all temporaries and arguments and *using* argument registers up to the arity of the called procedure.

We shall be considering the (somewhat contrived) clause:

```
p(N, T, V, S0, S) :-
        arg(N, T, A),
        A1=A,
        M is N+A1,
        p(A, V, S0, S1),
```

```
p(M, T, V, S1, S).
```

which previous phases have compiled to the following annotated list of instructions. The `heapmargin_...` instructions are omitted for simplicity. Each instruction is listed together with its corresponding use-def operators and live variable set:

Instruction	*Use-Def Ops*	*Live Variables*
(1)neck(5)	u(0)...u(4)	[0 1 2 3 4]
(2)get_variable(y(4),4)	u(4)	[0 1 2 3 4]
(3)get_variable(x(T3),3)	c(3,T3)	[0 1 2 3]
(4)get_variable(y(2),2)	u(2)	[0 1 2 3]
(5)get_variable(y(1),1)	u(1)	[0 1 2 3]
(6)get_variable(x(T4),0)	c(0,T4)	[0 1 2 3]
(7)function(arg,T0,0,1)	u(0) u(1) d(T0)	[0 1 2 3]
(8)put_value(x(T0),T2)	c(T0,T2)	[0 2 3 T0]
(9)function(+,T1,0,T2)	u(0) u(T2) d(T1)	[0 2 3 T0 T2]
(10)get_variable(y(0),T1)	u(T1)	[2 3 T0 T1]
(11)put_value(x(T0),0)	c(T0,0)	[2 3 T0]
(12)put_value(x(2),1)	c(2,1)	[0 2 3]
(13)put_value(x(3),2)	c(3,2)	[0 1 3]
(14)put_variable(y(3),3,_)	d(3)	[0 1 2]
(15)call(p/4,5,_)	u(0)...u(3)	[0 1 2 3]
(16)put_value(y(0),0)	d(0)	[ ]
(17)put_value(y(1),1)	d(1)	[0]
(18)put_value(y(2),2)	d(2)	[0 1]
(19)put_unsafe_value(y(3),3)	d(3)	[0 1 2]
(20)put_value(y(4),4)	d(4)	[0 1 2 3]
(21)execute(p/5)	u(0)...u(4)	[0 1 2 3 4]

where the source variables and internal temporaries have been assigned the object code names:

```
M  = y(0)              A  = x(T0)
T  = y(1)              <the result of is/2> = x(T1)
V  = y(2)              A1 = x(T2)
S1 = y(3)              S0 = x(T3)
S  = y(4)              N  = x(T4)
```

Notice that in instructions (7), (9), (12), and (13), the argument register cache has replaced uses of `x(T4)` by `x(0)`, `y(1)` by `x(1)`, `y(2)` by `x(2)`, and `x(T3)` by `x(3)`, respectively. Thus the temporaries `T3` and `T4` are not used anywhere, and instructions (3) and (6) represent useless copies.

The live variable sets can be notionally thought of as an *interference graph* as depicted in figure 3-1. Nodes represent temporaries and registers. A solid edge represents an *interference* which

occurs if two temporaries or registers are simultaneously live somewhere and contain different values. Since the latter criterium is in general undecidable, distinct temporaries and registers are assumed to always contain distinct values. This assumption will be relaxed later. A dotted edge between two nodes *i* and *j* corresponds to a copy `c(`*i*`,`*j*`)` which we would like to collapse:

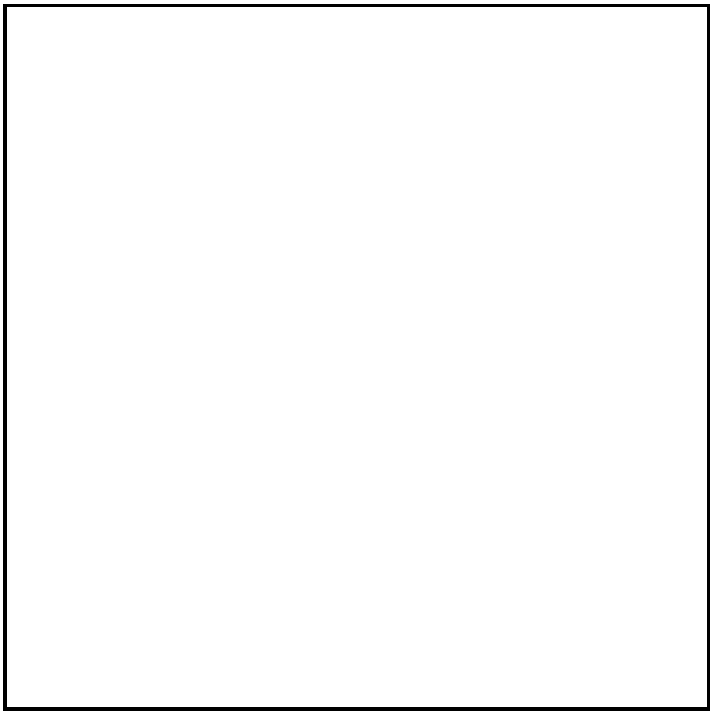

Figure 3-1: Interference graph for the first chunk

The task of the register allocator is to allocate registers to the remaining temporaries (`T0`, `T1`, `T2`), collapsing as many copies as possible, and maintaining the constraint that no pair of nodes connected by a solid edge be assigned the same register number.

3.6.2 Correct and Optimal Assignments

The compiler does not attempt to achieve a truly optimal assignment, since that problem is likely to be NP-complete. However, we can formulate two quality measures:

- It is desirable to transform as many `copy(`*i*`,`*j*`)` instructions as possible into no-ops, by assigning the same register number to *i* and *j*, as no-ops are deleted by the final editing phase.
- It is desirable to maximise the use of such argument registers that reside in machine registers. As this is highly implementation dependent, we simply assume that it is cheaper to access registers with low numbers than registers with high numbers.

We now formulate a correctness condition in terms of live variable sets. We then relax the condition by taking *aliasing* into account. Two temporaries or registers are aliases if they do hold the same value. Aliases are introduced by `copy` instructions.

> An assignment A for a chunk is correct if for each pair (i, j) of elements of any live variable set, $A(i) \neq A(j)$.

Thus an assignment is correct if it preserves the cardinality of all live variable sets, i.e. if it doesn't cause two interfering elements of the same set to become merged. Two elements becoming merged means that an argument register is holding two distinct values at once. In the code example of the preceding section, the temporary `T0` must not be assigned register number 0, since both `T0` and register 0 are live at entry to instructions (8–9), and are not aliases.

For each chunk we define a relation $\simeq$ such that $i \simeq j$ iff the chunk contains a `copy(`i`,`j`)` instruction and i and j are either temporary variables or argument registers denoting unique values (see section 3.4, page 13). We also define $\cong$ as the reflexive, symmetric and transitive closure of $\simeq$. In the code example of the preceding section, $\simeq$ contains the tuple `(T0,T2)` only, and the solid edge between those two nodes in the graph can safely be removed.

We can now formulate a relaxed correctness condition to express the fact that aliased elements may be merged:

> An assignment A for a chunk is correct if for each pair (i, j) of elements of any live variable set, $A(i) \neq A(j) \vee i \cong j$.

Note that there may be other correct assignments involving aliasing of reused argument registers, but we will not attempt to relax the correctness condition further. In the example of the preceding section, `T0` and `T2` may be merged even though they are both live at instruction (9), since they are certainly aliases. In terms of the interference graph, the nodes `T0` and `T2` do not actually interfere and so can be coalesced into one.

3.6.3 Implementation

As many copies as possible are collapsed first. Then the live variable sets are constructed and register numbers are assigned "on the fly" whenever an unallocated temporary is added to a live variable set. Three different methods are used; one for the first chunk of non-halting clauses, one for subsequent chunks of recursive clauses, and one for the single chunk of halting clauses.

The methods differ mainly in their treatment of copies where the operands are not in $\cong$, since each such copy requires consulting the live variable sets to see if the operands are simultaneously

live anywhere. Such copies can only occur in the first chunk of non-halting clauses; in the other contexts, all copies can be collapsed without any check.

The actual implementation treats aliases somewhat more efficiently than the algorithms described here. The $\cong$ relation is never actually constructed or consulted for the first chunk; instead, the previous phase (see section 3.5, page 25) collapses copies in the first chunk when it is safe to do so.

3.6.4 The First Chunk

The algorithm proceeds in three steps.

1. Each `copy(`*i*`,`*j*`)` instruction is considered, and *i* and *j* are equated if $i \cong j$.
2. Each remaining `copy(`*i*`,`*j*`)` instruction is considered, and *i* and *j* are equated if no live variable set contains both *i* and *j*. To check whether this is the case, we do not actually construct the live variable sets. Instead, we define a finite state automaton, as depicted in figure 3-2.

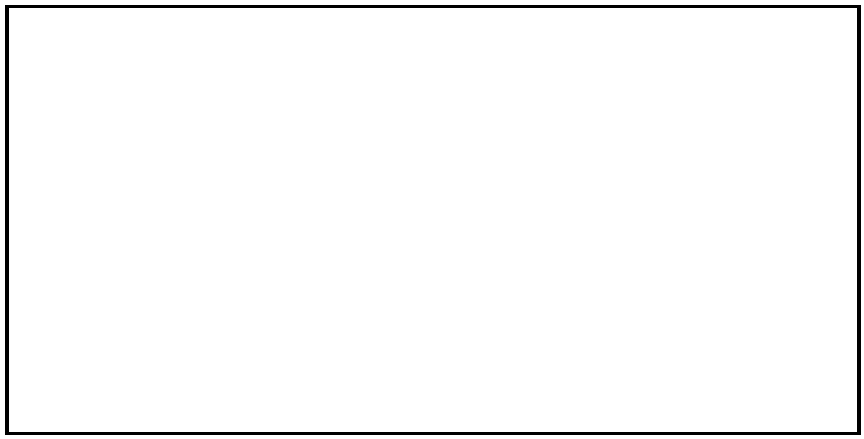

Figure 3-2: Finite state abstraction of live variable sets

where in- and outgoing arrows denote initial and final states and `Q` denotes a set which neither contains *i* nor *j*; `QI` denotes a set which contains *i* but not *j*; `QJ` denotes a set which contains *j* but not *i*; and `QIJ` denotes a set which contains both *i* and *j*.

We use the *ud* sequence in reverse order, in unison with the *f* function. State transitions occur when *i* or *j* is defined or used, but occurrences of other variables are ignored. We define the transition function *g* by the following table, where *x* is any variable distinct from *i* and *j*:

Operator	*Q*	*QI*	*QJ*
`d(i)`	`Q`	`Q`	`QJ`
`d(j)`	`Q`	`QI`	`Q`
`d(x)`	`Q`	`QI`	`QJ`

u(i)	QI	QI	QIJ
u(j)	QJ	QIJ	QJ
u(x)	Q	QI	QJ
c(i,i)	Q	QI	QJ
c(i,j)	Q	QI	QI
c(i,x)	QI	QI	QIJ
c(j,i)	Q	QJ	QJ
c(j,j)	Q	QI	QJ
c(j,x)	QJ	QIJ	QJ
c(x,i)	Q	Q	QJ
c(x,j)	Q	QI	Q
c(x,x)	Q	QI	QJ

---

and form a sequence of states as follows:

$$\mathrm{q}(m{+}1) = \mathtt{Q},$$
$$\mathrm{q}(k) = \mathrm{g}(\mathrm{q}(k{+}1), \mathrm{ud}(k), i, j), 0 \le k \le m.$$

Thus, i and j can be merged if the final state `QIJ` is not reached. To show that the final state is indeed reached if there is a set containing i and j, we must prove that g is a sound abstraction of f.

Theorem: For $0 \le k \le m$, $i \in \mathrm{lv}(k) \Rightarrow \mathrm{q}(k) = \mathtt{QI} \vee \mathrm{q}(k) = \mathtt{QIJ}$. For $0 \le k \le m$, $j \in \mathrm{lv}(k) \Rightarrow \mathrm{q}(k) = \mathtt{QJ} \vee \mathrm{q}(k) = \mathtt{QIJ}$.

Proof: by induction over k.

Since the finite automaton is an imperfect abstraction of the *lv* sequence, it may reach its final state even if there is no live variable set containing both i and j. To see this, consider the state transition rules for the operators `c(`i`,`x`)` and `c(`j`,`x`)`. Without knowing whether x is actually in the live variable set, we must assume that the operators represent useful copies, and so the finite automaton must do the respective state transition even if the copy was useless.

We have found this situation to be quite rare in practice.

3. The live variable sets are constructed, as described in the previous section. Whenever an uninstantiated set element T is encountered while walking the code in reverse order, it is assigned the smallest integer i such that i is not in the current set *and i is greater than or equal to the head arity*. If i is chosen less than the head arity, T could clash with an input argument in a live variable set which has not yet been computed. Whenever a `cut` instruction is encountered, one of its operands is instantiated to the smallest register number that is greater than all currently live registers.

Consider again the code example of the preceding section.

Step 1 is able to collapse instruction (8). Step 2 attempts to collapse instruction (11) but fails, since `T0` does interfere with register 0. Finally, step 3 assigns register 5 to `T0` (and to its alias `T2`) when it reaches instruction (11), and assigns register 6 to `T1` when it reaches instruction 10. Notice that it would have been correct (and better) to assign register 1 to `T0` and 0 to `T1` instead, but the

register allocator cannot detect that it is safe to do so until it knows the *whole* lifetime of `T0` and `T1`.

3.6.5 Subsequent Chunks

The algorithm for subsequent chunks of recursive clauses is considerably simplified, since all operands of copies are in $\cong$.

1. Each `copy(`i`,`j`)` instruction is considered, and i and j are equated if at least one of them is a temporary variable.
2. The live variable sets are constructed, as described in the previous section. Whenever an uninstantiated set element is encountered, it is assigned the smallest integer that is not in the set. Whenever a `cut` instruction is encountered, one of its operands is instantiated as in the First Chunk algorithm.

In this case, there are no input arguments which could clash with the registers allocated in step 2.

3.6.6 Single Chunk

The algorithm for the single chunk of halting clauses is a trivial variant of the Subsequent Chunk algorithm, in order to handle input arguments which could clash with the registers allocated in step 2.

1. Each `copy(`i`,`j`)` instruction is considered, and i and j are equated if at least one of them is a temporary variable.
2. The live variable sets are constructed, as described in the previous section. Whenever an uninstantiated set element is encountered, it is assigned the smallest integer i such that i is not in the set and i is greater than or equal to the head arity. Whenever a `cut` instruction is encountered, one of its operands is instantiated as in the First Chunk algorithm.

3.6.7 Complexity Analysis

We estimate the complexity of register allocation for a chunk for which there is a sequence $\mathrm{ud}(0) \ldots \mathrm{ud}(N)$ of use-def operators.

The complexity of collapsing variables in the Subsequent Chunk and Single Chunk algorithms is linear in N: each *c(i,j)* operator is treated in constant time. The complexity of collapsing variables

in the First Chunk algorithm is $O(N^2)$: each *c(i,j)* operator may require a traversal of the *ud* sequence.

The cost of computing a live variable set and of allocating an uninstantiated temporary is at most $O(K)$, where K is the maximum cardinality of any live variable set and depends on the complexity of the source code. Thus the total cost of constructing the live variable sets and of allocating the temporaries is $O(K \times N)$.

3.7 Final Editing

A final pass is made over the instruction sequence corresponding to a clause in the source program, or to a disjunct of a control structure which was broken out as an internal predicate. In this pass, a number of local changes are performed: deleting useless instructions, replacing certain instructions by cheaper versions, inserting instructions for (de)allocating environments, inserting and replacing instructions to support memory management, and filling in operands required to support the SRI model.

In particular, the `allocate1`, `allocate2`, and `deallocate` instructions must be inserted where appropriate. `allocate1` instructions are inserted as late as possible, i.e. just before the first procedure call or permanent variable reference in a clause, to increase the chance of avoiding the work done by `allocate1`.

Recall that the Aurora abstract machine insists that the environment be fully initialised at the first `call` for recursive clauses. The reason is to guarantee the consistency of the environment at all times that a garbage collection might occur. It is the responsibility of this phase to insert a sequence of `put_void` instruction followed by an `allocate2` instruction before the first `call` instruction to initialise permanent variables that have not yet been initialised. The `get_variable` and `unify_variable` instructions in subsequent chunks that would otherwise initialise the variables must also be changed to `get_first_value` and `unify_first_value` instructions, respectively. These latter instructions guarantee by trailing that no garbage pointers can be introduced into the environment by backtracking. As a compensation to the overheads incurred by this measure, `put_variable` instructions in subsequent chunks can be strength reduced to `put_value` instructions.

To support the SRI model, the *variable number* operand must be filled in for `put_variable` and `put_void` instructions initialising permanent variables, and the *active variable count* operand must be filled in for `call` instructions. Variable numbers are assigned by considering each permanent variable `y(`i`)` in turn, starting with $i = 0$. The first permanent variable to require a binding array slot is assigned the number 0, the next 1, and so on. The active variable count of a `call` is computed as the number of such permanent variables with binding array slots that are used after the `call`,

or as 1, if no such variables are used after the `call`. The reasons for not computing this operand as 0 are explained in [Carlsson and Szeredi 90].

The various editing operations are summarised in the table below.

`get_variable(x(`X`),`X`)`
`put_value(x(`X`),`X`)`
`put_variable(x(void),void,_)`
: are deleted

`put_variable(x(`X`),`X`,_)`
: is replaced by `put_void(x(`X`),_)`

`unify_variable(x(void))`
: is replaced by `unify_void`

`execute(true/0)`
: is replaced by `proceed`

`execute(fail/0)`
: is replaced by `fail`

`call(`P`,`n`,`m`)`
: is completed by instantiating m

`cut(x(-1),`L`,`Op`)`
: is replaced by `cut(`L`,`Op`)`

`get_variable(`N`,-1)`
: is replaced by `choice(`N`)`

`get_variable(y(`X`),`A`)`
: After the first `call`, this is replaced by `get_first_value(y(`X`),`A`)`

`put_variable(y(`X`),`A`,`m`)`
: After the first `call`, this is replaced by `put_value(y(`X`),`A`)`. Before the first `call`, the variable number m is instantiated.

`unify_variable(y(`X`))`
: After the first `call`, this is replaced by `unify_first_value(y(`X`))`.

`heapmargin_call(`I`,`A`)`
`heapmargin_exit(`I`)`
: are deleted if I is below a certain threshold

For example, the predicate

```
p(X) :-
        q(_, K),
        r(K),
        s(K, data(P), L),
```

```
t(L),
X=const,
u(P),
fail.
```

compiles to the instruction sequence displayed in the left hand column below. The right hand column displays the instruction sequence produced by the final editing phase:

Before editing	*Final code*
`heapmargin_call(1,1)`	
`neck(1)`	`neck(1)`
	`allocate1`
`get_variable(y(1),0)`	`get_variable(y(1),0)`
`put_variable(x(0),0,_)`	`put_void(x(0),_)`
`put_variable(y(3),1,_)`	`put_variable(y(3),1,2)`
	`put_void(y(0),0)`
	`put_void(y(2),1)`
	`allocate2`
`call(q/2,4,_)`	`call(q/2,4,3)`
`heapmargin_exit(0)`	
`put_value(y(3),0)`	`put_value(y(3),0)`
`call(r/1,4,_)`	`call(r/1,4,3)`
`heapmargin_exit(3)`	
`put_structure(data/1,1)`	`put_structure(data/1,1)`
`unify_variable(y(0))`	`unify_first_value(y(0))`
`put_unsafe_value(y(3),0)`	`put_unsafe_value(y(3),0)`
`put_variable(y(2),2)`	`put_value(y(2),2)`
`call(s/3,3,_)`	`call(s/3,3,2)`
`heapmargin_exit(1)`	
`put_unsafe_value(y(2),0)`	`put_unsafe_value(y(2),0)`
`call(t/1,2,_)`	`call(t/1,2,1)`
`heapmargin_exit(0)`	
`put_value(y(1),0)`	`put_value(y(1),0)`
`get_constant(const,0)`	`get_constant(const,0)`
`put_value(y(0),0)`	`put_value(y(0),0)`
`call(u/1,0,_)`	`call(u/1,0,1)`
`heapmargin_exit(0)`	
	`deallocate`
`execute(fail/0)`	`fail`

4. Optimisations

For iterative and recursive clauses, the register allocator is applied to two versions of the first chunk: one version containing the `neck` instruction and another version without it. Since the `neck` instruction constrains the register allocator, it can sometimes be unable to collapse copies in the "neck version", while those copies can be collapsed in the other version. It is arranged so that the "neck version" is the clause entry point when the clause is used when there are outstanding alternatives for the current procedure invocation, and the other version when there are not. Usually, a common suffix of the two code streams can be found, and the code is arranged so that the "neck version" branches into the common suffix while the "non-neck version" drops through into that suffix. The situation is depicted in figure 4-1.

Figure 4-1: Compiled code, its two entrypoints and common suffix

For halting clauses, only one version is produced. Although the `neck` instruction still constrains the register allocator, the constraints cannot prevent any copies from being collapsed.

Another optimisation is related to the indexed get instructions (see section 3.5, page 25). If a clause is entered by a procedure call (as opposed to by backtracking) and the first argument is not unbound, the indexing mechanism built into the Aurora emulator will already have performed any action that an indexed get instruction would perform, and the emulator simply skips any such instruction. This is achieved by splitting the above entrypoints into two subcases to be used in different circumstances. Thus there can be up to four different entrypoints per clause.

5. Comparison with Other Work

5.1 Other Prolog Compilers

Several compilers for the WAM have been produced and aspects of these implementation have been discussed in the literature. See e.g. [Newton 86] which describes the the Caltech IBM/370 implementation, [Van Roy 84] which describes the compiler of the Berkeley PLM, [Turk 85] which discusses issues in compiling to native code for the Syracuse University implementation, and [Tick 87] which discusses architectural requirements for efficient Prolog execution.

Optimisations of the WAM instruction set have been suggested by many researchers. See e.g. [Park and DeGroot], [Beer], and [Meier 88].

Several researchers have studied the problem of register allocation for the WAM. The overall structure of the Berkeley PLM compiler is similar to ours: code generation proper is separated from register allocation. The code generator emits worst case code, in the sense that it contains many copy instructions, anticipating that the register allocator will make many of them redundant. However, Van Roy's algorithm uses a backtracking-driven approach to register allocation, and hence can run in exponential time, whereas ours runs in quadratic time at worst. See section 3.6.7, page 35.

Debray describes in [Debray 86] three strategies for register allocation which intertwine code generation proper and register allocation: registers are allocated as each instruction is emitted, and the register allocator may itself cause copy instructions to be emitted. His algorithms require that the last occurrence of each temporary variable T be specially annotated in the abstract syntax tree, in order to free the register occupied by T at its last occurrence. They further require that sets of preferable or unsuitable argument registers be computed for each T before code generation can begin. The three strategies differ in their treatment of *conflicts*, i.e. situations where different temporary variables compete for the same argument register:

Conflict Avoidance
: Registers are allocated so that conflicts never appear. This sometimes leads to wasted work, as copy instructions may be emitted to avoid conflicts before any "useful" instructions appear.

Conflict Resolution
: Conflicts are resolved as they occur. Unfortunately, conflicts may cascade which leads to temporaries having to be copied from one register to another many times.

Mixed Conflict Resolution and Avoidance

This is a hybrid of the other two. Conflicts are resolved as they occur, but in a way so that resolving them cannot cause new conflicts.

Our algorithm is of the Conflict Avoidance type (the register allocator never inserts new copy instructions), but tries to collapse as many copies as possible before register allocation proper begins. The negative effects of avoiding conflicts overly eagerly are mitigated by our heuristic compilation order for head and body arguments (see section 3.4.3, page 18, see section 3.4.4, page 19).

An algorithm which is a refinement of Debray's hybrid algorithm, augmented by an *adaptable unification order*, is presented in [Janssens et al. 88]. The algorithm distinguishes between three types of head and body arguments:

1. Arguments that do not contain temporary variables.
2. Arguments that are temporary variables.
3. Compound terms that contain temporary variables.

and compile the arguments of body goals in the order 3—2—1, and head arguments in the order 1—2—3. This compilation order is similar to but more ambitious than our heuristic order.

The [Janssens et al. 88] paper contains several examples comparing code generated by Debray's hybrid method and by theirs. We present below some of the examples with code generated by our compiler. The `neck` instruction is not shown in these examples.

1. This example shows the advantage of not compiling head arguments strictly from left to right. Janssens' method and ours produce identical code (modulo our specialised `get_structure_x0` instruction); Debray's method produces one more instruction.

```
del(t(L,X,R), X, t(L,Y,R1)) :-
        delmin(R,Y,R1).
```

Debray's method	*Janssens' method and Aurora*
`get_structure(t/3,0)`	`get_structure(t/3,0)`
`unify_variable(x(3))`	`unify_variable(x(3))`
`unify_variable(x(4))`	`unify_local_value(x(1))`
`unify_variable(x(0))`	`unify_variable(x(0))`
`get_value(x(4),x(2))`	
`get_structure(t/3,2)`	`get_structure(t/3,2)`
`unify_value(x(3))`	`unify_value(x(3))`
`unify_variable(x(1))`	`unify_variable(x(1))`
`unify_variable(x(2))`	`unify_variable(x(2))`
`execute(delmin/3)`	`execute(delmin/3)`

2. In this example, the more ambitious reordering of Janssens' method produces one less copy than our method:

```
p(T, U, a) :- q(T, b, f(U)).

Janssens' method                Aurora

get_constant(a,2)               get_constant(a,2)
                                get_variable(x(3),1)
put_structure(f/1,2)            put_constant(b,1)
unify_local_value(x(1))         put_structure(f/1,2)
put_constant(b,1)               unify_local_value(x(3))
execute(q/3)                    execute(q/3)
```

3. For the `p/4` clause below, Janssens' more ambitious conflict handling method produces better code than Debray's method. No compiled code for the `p/5` clause was available for Debray's and Janssens' methods.

   This example also shows that our simple heuristic of compiling variables in the head from right to left and variables in body goals from left to right suffices for achieving optimal code in clauses where several arguments are passed from the head to the body goal, in the same order but in different argument position.

```
p(T, U, V, W) :- q(a, T, U, V, W).

Debray's method          Janssens' method and Aurora

get_variable(x(5),0)     get_variable(x(4),3)
put_constant(a,0)        get_variable(x(3),2)
get_variable(x(6),1)     get_variable(x(2),1)
put_value(x(5),1)        get_variable(x(1),0)
get_variable(x(5),2)     put_constant(a,0)
put_value(x(6),2)        execute(q/5)
get_variable(x(4),3)
put_value(x(5),3)
execute(q/5)
```

```
p(a, T, U, V, W) :- q(T, U, V, W).

Aurora

get_constant_x0(a)
put_value(x(1),0)
put_value(x(2),1)
put_value(x(3),2)
put_value(x(4),3)
execute(p/4)
```

Finally, a rigorous approach to achieving optimal code by using a graph representation to resolve register conflicts in an optimal way is presented in [Abe et al. 86].

5.2 Graph Colouring

In the compiler literature, register allocation is often treated as the problem of colouring an interference graph [Chaitin et al. 81], such as the one shown earlier (see section 3.6.1, page 28), such that no pair of nodes connected by a solid edge may be assigned the same colour, where a colour corresponds to a register number. Evidently this approach is closely related to the method used in our compiler: intermediate code is generated with symbolic names instead of actual registers, an interference graph is computed, and the nodes of the graph are coloured. The graph colouring methods have treated the aliasing problem much the same way as we have: pairs of nodes representing aliases are coalesced into single nodes.

Instead of first computing the interference graph and then colouring it, our compiler notionally colours each node "on the fly" while constructing the graph whenever it encounters a temporary by assigning to it a register number which cannot interfere with the temporary. The advantage of our method is that the colouring does not require a separate pass over the graph, and that the whole graph does not have to be maintained as a data structure. The disadvantage of our method is that it is overly conservative: a suboptimal register number is often chosen (cf. step 3 of the First Chunk register allocation algorithm).

The general graph colouring algorithms often have a limited numbers of colours available. It is well known that the problem of n-colouring a graph is NP-complete [Garey and Johnson 88]. However, our compiler simply assumes that an unlimited number of registers are available.

6. Conclusions

We have described algorithms for a compiler which compiles one Prolog clause at a time into an abstract instruction set supporting or-parallel execution. The instruction set is based on the sequential Warren Abstract Machine with extensions for full Prolog, shallow backtracking, memory management, and for the SRI model of or-parallel execution of Prolog.

Most of the described algorithms apply to compilation of sequential Prolog programs. The extensions introduced to support or-parallelism are minor. The compiler has been implemented in Prolog, and is used in the Aurora Or-Parallel Prolog system, where it translates one clause at a time into a sequence of symbolic bytecode instructions plus indexing and pruning information. The Aurora compiler is an extension of the compiler used in the sequential SICStus Prolog. The same compiler has been extended by others for restricted and-parallelism [Hermenegildo 88] and for inserting instructions into compiled code to support the gathering of profiling data [Gorlick and Kesselman 87].

The compiler does not specially optimise disjunctions, but treats these as calls to anonymous predicates which are compiled recursively. Although some opportunities for optimisation are sacrificed, this approach greatly simplifies other parts of the compiler.

Register allocation is done in a fashion similar to colouring an interference graph. Code generation proper is done with temporaries left unallocated, and uses heuristics for finding a compilation order which minimises the number of register-register copies. After such copies have been coalesced where possible, register allocation is performed in a single pass over the intermediate code. The register allocation method is quite general, and it should be possible to adapt it for use in other languages.

A final editing phase performs local changes and optimisations, producing the final abstract code. The final editing includes assinging values to operands which have been added to certain instructions to support compile-time allocation of binding array offsets for permanent variables (generalised environment trimming).

Acknowledgements

The author is indebted to Seif Haridi, Sverker Janson, Dan Sahlin, Martin Nilsson, Ewing Lusk, Vítor Santos Costa, Manuel Hermenegildo, and to members of the Logic Programming Systems Laboratory at SICS for careful reading and valuable comments on drafts of this report.

References

[Abe et al. 86]
S. Abe, K. Kurosawa, K. Kiriyama, *A New Optimisation Technique for a Prolog Compiler*, Conference on Computers, pp. 241–245, IEEE Computer Society, 1986.

[Aho et al. 85]
A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers—Principles, Techniques, and Tools*, Addison-Wesley, 1985.

[Appleby et al. 88]
K. Appleby, M. Carlsson, S. Haridi, D. Sahlin, *Garbage Collection for Prolog Based on WAM*, Communications of the ACM vol. 31 no. 6 pp. 719–741, 1988.

[Beer] J. Beer, *A Critique of Warren's Abstract Instruction Set*, internal report, GMD–FIRST, TU Berlin.

[Carlsson 87]
M. Carlsson, *Freeze, Indexing and Other Implementation Issues in the WAM*, Proc. Fourth International Conference on Logic Programming, pp. 40–58, MIT Press, 1987.

[Carlsson 89]
M. Carlsson, *On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog*, Proc. Sixth International Conference on Logic Programming, pp. 3–16, MIT Press, 1989.

[Carlsson and Szeredi 90]
M. Carlsson, P. Szeredi, *The Aurora Abstract Machine and its Emulator*, SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[Chaitin et al. 81]
G.J. Chaitin et al. , *Register Allocation via Coloring*, Computer Languages vol. 6 pp. 47–57, 1981.

[Debray 86]
S.K. Debray, *Register Allocation in a Prolog Machine*, Proc. Symposium on Logic Programming, pp. 267–275, IEEE Computer Society, 1986.

[Gorlick and Kesselman 87]
M.M. Gorlick, C.F. Kesselman, *Timing Prolog Programs Without Clocks*, Proc. Symposium on Logic Programming, pp. 426–432, IEEE Computer Society, 1987.

[Hermenegildo 88]
M.V. Hermenegildo, *Independent/Restricted AND-parallel Prolog and its Architecture*, Kluwer Academic Publishers, 1988.

[Garey and Johnson 88]
M.R. Garey, D.S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, New York, 1979.

[Janssens et al. 88]
G. Janssens et al. , *Improving the Register Allocation in WAM by Reordering Unification*, Proc. Fifth International Conference on Logic Programming, pp. 1388–1402, MIT Press, 1988.

[Lusk et al. 88]
E. Lusk, D.H.D. Warren, S. Haridi et al. , *The Aurora Or-Parallel System*, New Generation Computing vol. 7 nos. 2–3 pp. 243–271, 1990.

Proc. International Conference on Fifth Generation Computer Systems, pp. 819–830, ICOT, Tokyo, 1988.

[Meier 88] M. Meier, *Benchmarking of Prolog Procedures for Indexing Purposes*, Proc. International Conference on Fifth Generation Computer Systems, pp. 800–807, ICOT, Tokyo, 1988.

[Newton 86]
M.O. Newton, *A High Performance Implementatio of Prolog*, Report No. 4234:TR:86, Compputer Science Department, California Institute of Technology, 1986.

[Park and DeGroot]
S-K. Park, D. DeGroot, *Clause-Level Optimization of Abstract Prolog Instruction Set*, internal report, IBM Thomas J. Watson Research Center.

[SICS 88] *SICStus Prolog User's Manual*, SICS Research Report R88007B, 1988.

[Tick 87] E. Tick, *Studies in Prolog Arhitectures*, Ph.D. dissertation, Technical Report No. CSL-TR-87-329, Department of Electrical Engineering and Computer Science, Stanford University, 1987.

[Turk 85] A. K. Turk, *Compiler Optimizations for the WAM*, School of Computer and Information Science, Syracuse University, Technical Report CIS-85-6, November 1985.

[Van Roy 84]
P. Van Roy, *A Prolog Compiler for the PLM*, Report No. UCB/CSD 84/203, Computer Science Division (EECS), University of California at Berkeley, 1984.

[Warren 83]
D.H.D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, 1983.

[Warren 87]
D.H.D. Warren, *The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues*, invited talk, Proc. Symposium on Logic Programming, pp. 92–102, IEEE Computer Society, 1987.

Appendix I: Synopsis of the WAM

The Warren Abstract Machine, or WAM, was invented in 1983 by D.H.D. Warren, and represented a major breakthrough in Prolog compiler technology. It is defined as an instruction set and several data areas that the instructions operate on [Warren 83]. The instructions are oriented towards the source language rather than towards the target machine. Thus an abstract instruction usually corresponds to a source program symbol but may involve a large, even unbounded, number of machine operations.

The WAM has since become generally accepted as the standard Prolog implementation technique and has been realised by emulation of bytecoded abstract instructions, by compilation into native machine instructions, and by emulation in firmware and hardware. We shall not contemplate these techniques further.

Several researchers have described optimised or extended versions of the WAM. The rest of this appendix contains a condensed description of a somewhat simplified WAM. For instance, we do not treat all instructions, nor their complete semantics. See chapter 2, page 3 for some necessary terminology.

Data Areas

The data areas are the *code area*, containing the program itself, and three areas operated as stacks: The *local stack* contains information pertaining to backtracking and recursive procedure invocations. The *global stack* contains structures, lists, and value cells created by Prolog execution. The *trail stack* contains references to conditionally bound variables, i.e. it records those bindings that have to be undone upon backtracking.

Abstract Machine Registers

The current computation state is defined by the contents of the abstract machine registers. The principal such registers are:

`P`	Program pointer (to the code area).
`CP`	Continuation program pointer (to the code area).
`E`	Current environment (on the local stack).
`B`	Current choicepoint (on the local stack).
`TR`	Trail stack pointer.
`H`	Global stack pointer.
`S`	Structure pointer (to the global stack).
`X[]`	Argument register bank.

The Global Stack

An important concept is that of a *value cell*. A value cell is a machine word which stores a Prolog term, represented as a bit string with a tag part and a value part. The global stack consists exclusively of value cells. There are value cells on the local stack as well. The following rules govern how value cells may refer to one another:

- A global value cell may only refer to another global value cell.
- A local value cell may refer to an earlier local value cell or to a global value cell.

The tag distinguishes the type of the term, the main types being *references*, *compound terms* (*structures* and *lists*), and *constants*. An *unbound variable* is represented as a reference to itself. A value cell tagged as a variable but not pointing to itself represents a *bound variable*. Structures are created by explicitly copying the functor and arguments into consecutive value cells on the global stack. The value part of the functor value cell encodes its function symbol and arity. Lists are created similarly, except no functor needs to be stored. This technique for creating structures and lists is known as *structure copying*. For constants, the value field contains a symbol table index rather than a pointer.

A procedure known as *dereferencing* follows a chain of bound variables until an unbound variable or a non-variable is encountered. An unbound variable is *global* is the value cell is located on the global stack; otherwise, it is *local*. A procedure known as *globalising* creates a new global variable and binds a local variable to it, ensuring that the local variable henceforth dereferences to the global stack.

The Local Stack

The local stack contains two kinds of structures: *environments* and *choicepoints*. An environment represents a list of goals still to be executed. It consists of a number of value cells for variables occurring in the body of a clause plus a pointer into the body of a continuation clause and its environment. A local stack variable may be unbound, in which case it points to itself just like unbound global stack variables. Unbound variables on the local stack save global stack storage but complicate the instruction set somewhat. We ignore these complications in this description.

Environments are only needed for recursive clauses, and are created when entering such clauses. They are (logically) discarded before executing the last body goal.

A choicepoint is created when entering a procedure which has more than one clause that could match the goal. It stores the values of the `H`, `TR`, `B`, `CP`, `E` registers, a pointer to the alternative clause, and the argument of the procedure. When no more alternatives remain, the choicepoint is discarded.

Notice that the top of local stack is not recorded explicitly. Instead, when an environment or a choicepoint is created, it is placed at the next available location, computed as:

$$\max(\texttt{B}, \texttt{E}+\text{env\_size}(\texttt{CP}))$$

where `env_size(CP)` refers to the second operand of the `call` instruction pointed to by the continuation program counter, as this indicates how many permanent variables the continuation will use. Thus a choicepoint "freezes" all existing structures on the local stack, preventing them from being physically deallocated or written over.

The Trail Stack

This area records *conditional* variable bindings. A variable needs to be bound conditionally if and only if the variable is older than the latest choicepoint. Address comparison is used in order to determine this. For global stack variables, the WAM compares the address of the variable with the `H` field of the current choicepoint. For local stack variables, the address of the variable is compared with the value of the `B` register. Upon backtracking, entries are simply popped from the trail stack and the bound variables are reset to unbound.

Treatment of Variables

Each source variable is statically classified as *permanent* if it is used in more than one chunk, or as *temporary* otherwise. Thus permanent variables have to survive procedure calls, whereas temporary variables temporarily hold data during unification and are used for passing arguments in procedure calls. Procedures do not need to preserve temporary variables. Permanent variables are written as `y(`i`)` in the abstract instructions, where i is an offset (≥ 0) in the value cell vector of the current environment. Temporary variables are written as `x(`i`)`, where i is an offset (≥ 0) in the argument register bank.

Permanent variables are allocated in such a way that they can be discarded as early as possible: the permanent variable whose last occurrence is earliest will get the slot with the highest location. Thus environments shrink by zero or more slots for each body goal, releasing local stack space piecewise rather than an environment at a time. This optimisation is called *environment trimming* and generalises tail recursion optimisation.

Consider, for example, the clause

```
p(A, B, S0, S) :-
        q(A, B, C, C),
        r(S0, S1),
        r(S1, S).
```

where `A`, `B`, and `C` are temporary and all other variables are permanent, allocated as:

```
y(0) = S1
y(1) = S
y(2) = S0
```

A somewhat less restrictive definition of permanent variables is used in [Warren 83].

The Instruction Set

Prolog programs compile into sequences of instructions, approximately one instruction per Prolog symbol. Instructions can take up to two operands, identifying variables (V), integers (i), constants (C), functors (F), and procedures (P). The instruction set for Prolog clauses is classified into *get*, *put*, *unify*, and *procedural* instructions. The compiled clauses that make up a Prolog procedure are linked together by *indexing* instructions, defined in [Warren 83] but not discussed in this report. We shall consider the following example when discussing the instruction set:

```
#1:     concatenate([],L,L).
#2:     concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

A naive assignment of temporary variables in clause #2 is to reserve `x(0)`–`x(2)` for the arguments and `x(3)`–`x(6)` for `X`, `L1`, `L2`, and `L3`, respectively. There are no permanent variables.

Get Instructions

Get instructions correspond to head arguments. They match against the procedure's arguments, passed in argument registers:

`get_variable(V,i)`
: V is assigned the value of `x(`i`)`.

`get_value(V,i)`
: The values of V and `x(`i`)` are unified.

`get_constant(C,i)`
: The value of `x(`i`)` is unified with the constant C.

`get_nil(i)`
: The value of `x(`i`)` is unified with the empty list.

`get_structure(F,i)`
: The value of `x(`i`)` is unified with the structure `F(...)`

`get_list(i)`
: The value of `x(`i`)` is unified with a list.

Thus the suffix (`constant`, `list`, etc.) suggests what the argument should match. The `variable` suffix stands for the first occurrence of a variable; `value` is used for subsequent occurrences.

The head arguments of clause #2 compile to:

```
get_list(0)             % concatenate([
unify_variable(x(3))    %              X|
unify_variable(x(4))    %                L1],
get_variable(x(5),1)    %             L2,
get_list(2)             %             [
unify_value(x(3))       %              X|
unify_variable(x(6))    %                L3])
```

Put Instructions

Put instructions correspond to body arguments. They load argument registers:

`put_variable(V,i)`
: V and `x(`i`)` are assigned a new variable.

`put_void(i)`
: A special case of the above item, used when a body argument is written as a singleton variable. `x(`i`)` as assigned a new variable.

`put_value(V,i)`
: `x(`i`)` is assigned the value of V.

`put_constant(C,i)`
: `x(`i`)` is assigned the constant C.

`put_nil(i)`
: `x(`i`)` is assigned the empty list.

`put_structure(F,i)`
: `x(`i`)` is assigned the structure `F(...)`.

`put_list(i)`
: `x(`i`)` is assigned a list.

The arguments of the goal of clause #2 compile to:

```
put_value(x(4),0)       % concatenate(L1,
put_value(x(5),1)       %             L2,
put_value(x(6),2)       %             L3)
```

Unify Instructions

Unify instructions correspond to arguments of a list or structure, and operate in *read mode* or *write mode*. If `x(`i`)` contains a structure with functor `F/n`, the instruction `get_structure(F/n,i)` will put WAM in read mode and set the `S` register pointing at the arguments. Following the get instruction, n unify instructions will match subterms accessed via the `S` register, running in read mode. If instead `x(`i`)` contains an uninstantiated variable, the get instruction will bind this variable to a structure which is about to be created, place `F/n` at the top of the global stack, and put WAM in write mode. The unify instructions will fill in the subterms, running in write mode.

The `get_list` instruction operates similarly. The `put_structure` and `put_list` instructions always put WAM in write mode.

Read mode semantics:

`unify_variable(V)`
: V is assigned the next subterm.

`unify_void`
: A special case of the above item, used when a subterm is written as a singleton variable. The next subterm is ignored.

`unify_value(V)`
: The value of V is unified with the next subterm.

`unify_constant(C)`
: The next subterm is unified with the constant C.

`unify_nil`
: The next subterm is unified with the empty list.

Write mode semantics:

`unify_variable(V)`
: A new variable is stored in V and as the next subterm.

`unify_void`
: A special case of the above item, used when a subterm is written as a singleton variable. A new variable is stored as the next subterm.

`unify_value(V)`
: The value of V is stored as the next subterm.

`unify_constant(C)`
: The constant C is stored as the next subterm.

`unify_nil`
: The empty list is stored as the next subterm.

Note that there are no unify instructions for lists or structures, as they would require extensions to the WAM data areas or registers. Instead, structures are "flattened" by introducing new temporary variables. For instance, a head '`p([d(0,a)])`' could compile to:

```
get_list(0)             % p([
unify_variable(j)       %    T1
unify_nil               %      ]) :-
get_structure(d/2,j)    % T1 = d(
unify_constant(0)       %        0,
unify_constant(a)       %        a)
```

whereas as a goal, it could compile to:

```
put_structure(d/2,j)    % T1 = d(
unify_constant(0)       %        0,
unify_constant(a)       %        a),
put_list(0)             % p([
unify_value(j)          %    T1
unify_nil               %      ])
```

Procedural Instructions

These instructions correspond to the head and goals of a clause. They deal with control transfer and environment handling:

`proceed` Branch to the continuation program pointer.

`execute(`*Q*`)`
: Branch to the procedure *Q*.

`call(`*Q*`,`*i*`)`
: Set the continuation program pointer at the next instruction, then branch to the procedure *Q*. The continuation will use *i* permanent variables of the current environment.

`allocate` Establish an environment.

`deallocate`
: Logically discard an environment.

Clause compilation uses one of the following patterns, where *I*, *J*, and *K* stand for sequences of inline goals, *P*, *Q*, and *R* stand for procedure calls, and *i* and *j* are integers indicating the size of the part of the current environment which is active after the respective `call` instruction:

halting clauses
: A clause '`H :- I.`' compiles to

```
get args of H
do inline goals I
proceed
```

iterative clauses
: A clause '`H :- I, P.`' compiles to

```
get args of H
do inline goals I
execute(P)
```

recursive clauses

A clause '`H :- I, P, J, Q, L, R.`' compiles to

`allocate`
get args of `H`
do inline goals `I`
put args of `P`
`call(P,i)`
do inline goals `J`
put args of `Q`
`call(Q,j)`
do inline goals `L`
put args of `R`
`deallocate`
`execute(R)`

Appendix II: Instruction Set

This chapter is a synopsis of the Aurora instruction set, which is an extension of the WAM instruction set presented in Appendix I. The extensions are explained in a previous section (see section 2.2, page 4). Although the format of most WAM instructions remains unchanged, it is worth noting that the `put_variable` and `call` instructions have acquired extra operands to support generalised environment trimming. The full semantics of the instructions is described in [Carlsson and Szeredi 90].

In the sections below, "first occurrence" and "subsequent occurrence" refer to compilation order, which may be somewhat different from textual order since head and goal arguments can be reordered. The notation *Vn* stands for a temporary or permanent variable.

Put Instructions

These instructions correspond to body arguments. They prepare arguments for the next procedure call or inline goal.

`put_void(x(`*i*`),_)`
: This represents an *i*th goal argument that is a singleton variable.

`put_void(y(`*n*`),`*m*`)`
: This represents a permanent variable `y(`*n*`)` that does not occur in the first chunk, and so must be explicitly initialised there. The variable is the *m*th variable in the current environment to be explicitly initialised.

`put_variable(x(`*n*`),`*i*`,_)`
: This represents an *i*th goal argument that is the first occurrence of the temporary variable `x(`*n*`)`.

`put_variable(y(`*n*`),`*i*`,`*m*`)`
: This represents an *i*th goal argument that is the first occurrence of the permanent variable `y(`*n*`)`, if it occurs in the first chunk. The variable is the *m*th variable in the current environment to be explicitly initialised.

`put_value(`*Vn*`,`*i*`)`
: This represents an *i*th goal argument that is a subsequent occurrence of the variable *Vn* which cannot point (even before dereferencing) to a portion of the stack which is about to be deallocated.

`put_unsafe_value(`*Vn*`,`*i*`)`
: This represents an *i*th goal argument that is a subsequent occurrence of the variable *Vn* which might point to a portion of the stack which is about to be deallocated, i.e. it might need globalising.

`put_constant(`*C*`,`*i*`)`
: This represents an *i*th goal argument that is the constant *C*.

`put_nil(`*i*`)`
: This represents an *i*th goal argument that is the empty list.

`put_structure(`*F*`,`*i*`)`
: This represents an *i*th goal argument that is a structure whose functor is *F*. The instruction is followed by a sequence of `unify` instructions.

`put_list(`*i*`)`
: This represents an *i*th goal argument that is a list. The instruction is followed by two `unify` instructions.

Get Instructions

These instructions correspond to head arguments and to certain inline goal arguments. They match head arguments against actual arguments and values computed by inline goals against other terms.

`get_variable(x(`*n*`),`*i*`)`
: This represents an *i*th head argument or the result of an inline goal that is the first occurrence of the temporary variable `x(`*n*`)`.

`get_variable(y(`*n*`),`*i*`)`
: This represents an *i*th head argument or the result of an inline goal in the first chunk that is the first occurrence of the permanent variable `y(`*n*`)`.

`get_first_value(y(`*n*`),`*i*`)`
: This represents the result of an inline goal after the first chunk that is the first occurrence of the permanent variable `y(`*n*`)`.

`get_value(`*Vn*`,`*i*`)`
: This represents an *i*th head argument or the result of an inline goal that is a subsequent occurrence of the temporary variable *Vn*.

`get_constant(`*C*`,`*i*`)`
: This represents an *i*th head argument or the result of an inline goal that is the constant *C*.

`get_nil(`*i*`)`
: This represents an *i*th head argument or the result of an inline goal that is the empty list.

`get_structure(`*F*`,`*i*`)`
: This represents an *i*th head argument or the result of an inline goal that is a structure with the functor *F*. The instruction is followed by a sequence of `unify` instructions.

`get_list(`*i*`)`

This represents an *i*th head argument or the result of an inline goal that is a list. The instruction is followed by two `unify` instructions.

Unify Instructions

These instructions correspond to arguments of compound terms. Each one has two modes of operation, read and write. In read mode, the structure pointer points to the next structure argument that the instruction should match. In write mode, the next structure argument is written to the top of the global stack.

`unify_void`

This represents a structure argument that is a singleton variable.

`unify_variable(x(`*n*`))`

This represents a structure argument that is the first occurrence of the temporary variable `x(`*n*`)`.

`unify_variable(y(`*n*`))`

This represents a structure argument that is the first occurrence of the permanent variable `y(`*n*`)`, occurring in the first chunk.

`unify_first_value(y(`*n*`))`

This represents a structure argument that is the first occurrence of the permanent variable `y(`*n*`)`, occurring after the first chunk.

`unify_value(`*Vn*`)`

This represents a structure argument that is a subsequent occurrence of the variable *Vn* which cannot (even before dereferencing) be pointing to the stack.

`unify_local_value(`*Vn*`)`

This represents a structure argument that is a subsequent occurrence of the variable *Vn* which could be pointing to the stack, i.e. it could need globalising.

`unify_constant(`*C*`)`

This represents a structure argument that is the constant *C*.

`unify_nil`

This represents a structure argument that is the empty list.

`unify_structure(`*F*`)`

This represents a last structure argument that is a structure with the functor *F*.

`unify_list`

This represents a last structure argument that is a list.

Procedural Instructions

These instructions correspond to the head and goals of a clause. They deal with recursive procedure calls and the data structures necessary for them.

`allocate1`
: This appears before the first reference to a permanent variable in a recursive clause. It must be matched by a `deallocate` instruction. Space for a new environment is allocated on the stack.

`allocate2`
: This appears just before the first procedure call of a recursive clause. The new environment is completed.

`deallocate`
: This appears after computing the arguments of the last goal in a recursive clause. The current environment is deallocated, updating the current environment and continuation WAM registers.

`call(`*Pred*`,`*k*`,`*m*`)`
: This corresponds to a procedure call that does not terminate a clause. The argument *k* is the active environment size after the call; *m* is the active variable count after the call; *Pred* is the predicate being called.

`execute(`*Pred*`)`
: This corresponds to a procedure call that terminates a clause. *Pred* denotes a predicate.

`proceed` This terminates a halting clause.

`fail` This causes backtracking to the latest choicepoint.

Indexing Instructions

These instructions interact with the clause indexing mechanism which in the emulator in built into the procedure call mechanism. They also deal with choicepoint handling.

`get_constant_x0(`*C*`)`
: This represents a first head argument that is the constant *C*. `x(0)` is already dereferenced and is either unbound or bound to *C*.

`get_nil_x0`
: This represents a first head argument that is the empty list. `x(0)` is already dereferenced and is either unbound or bound to the empty list.

`get_structure_x0(`*F*`)`
: This represents a first head argument that is a structure whose functor is *F*. `x(0)` is already dereferenced and is either unbound or bound to a structure whose functor is *F*. The instruction is followed by a sequence of `unify` instructions.

`get_list_x0`

This represents a first head argument that is a list. `x(0)` is already dereferenced and is either unbound or bound to a list. The instruction is followed by two `unify` instructions.

`branch(`*Offset*`)`

This occurs at a point in a clause after a `neck(_)` instruction and indicates a branch from the nondeterministic head code stream to a shared body code stream.

`neck(`*N*`)` This represents a point in a clause between the head and the first procedure call, where it is appropriate to decide whether a choicepoint needs to be allocated or updated. The clause belongs to a predicate of *N* arguments.

Utility Instructions

These instructions deal with pruning operators, inline goals and global stack overflow checks.

`choice(`*Vn*`)`

This represents the presence of a pruning operator, either an explicit one or a cut implictly introduced by a control structure. A pointer to the choicepoint that the operator should cut back to is stored in the variable *Vn*.

`cut(`*Vn*`,`*L*`,`*Op*`)`

This represents a pruning operator, either an explicit one or a cut implictly introduced by a control structure. The operator resets the current choicepoint from the variable *Vn*. Argument registers `x(0)`, ..., `x(`*L*`)-1` are live at this point.

`cut(`*L*`,`*Op*`)`

Equivalent to `cut(`*Vn*`,`*L*`,`*Op*`)` if the pruning operator occurs in the same chunk as the matching `choice(`*Vn*`)` instruction. If there are no more uses of *Vn*, the `choice` instruction can be deleted. Argument registers `x(0)`, ..., `x(`*L*`)-1` are live at this point.

`function(`*Name*`,`*k*`,`*i*`)`
`function(`*Name*`,`*k*`,`*i*`,`*j*`)`

These correspond to an application of a builtin function `x(`*k*`)` = *Name*(`x(`*i*`)`[`,x(`*j*`)`]). All other argument registers are preserved.

`builtin(`*Name*`,`*i*`)`
`builtin(`*Name*`,`*i*`,`*j*`)`
`builtin(`*Name*`,`*i*`,`*j*`,`*k*`)`

These correspond to an application of a builtin procedure *Name*(`x(`*i*`[,x(`*j*`)[,x(`*k*`)]]`)). The builtin procedure preserves all argument registers and must not create choicepoints.

`heapmargin_call(`*i*`,`*j*`)`

This represents the fact that a clause may require that at least *i* cells remain in the global stack. The clause belongs to a predicate with arity *j*. If less than *i* cells remain in the global stack, a global stack overflow routine is invoked.

`heapmargin_exit(`*i*`)`

This represents the fact that a continuation may require that at least *i* cells remain in the global stack. If less than *i* cells remain in the global stack, a global stack overflow routine is invoked.

Appendix III: Instructions and Use-Def Operators

Each instruction can be abstracted into a list of operators which denote uses or definitions of the operands. Such operators are useful in register allcation. See section 3.6.1, page 28 for a description of the semantics of the operators. Certain instructions introduced by the final editing phase are not present in this table.

Instruction	*Operators*
`neck(N)`	`u(0) u(1) ... u(N-1)`
`execute(P/N)`	`u(0) u(1) ... u(N-1)`
`call(P/N,K,M)`	`u(0) u(1) ... u(N-1)`
`cut(x(A),_,_)`	`u(A)`
`builtin(_,X)`	`u(X)`
`builtin(_,X,Y)`	`u(X) u(Y)`
`builtin(_,X,Y,Z)`	`u(X) u(Y) u(Z)`
`function(_,V,X)`	`u(X) d(V)`
`function(_,V,X,Y)`	`u(X) u(Y) d(V)`
`get_variable(x(V),A)`	`c(A,V)`
`get_variable(y(_),A)`	`u(A)`
`get_value(x(V),A)`	`u(V) u(A)`
`get_value(y(_),A)`	`u(A)`
`get_constant(_,A)`	`u(A)`
`get_nil(A)`	`u(A)`
`get_structure(_,A)`	`u(A)`
`get_list(A)`	`u(A)`
`put_variable(x(V),A,_)`	`d(V) d(A)`
`put_variable(y(_),A,_)`	`d(A)`
`put_value(x(V),A)`	`c(V,A)`
`put_value(y(_),A)`	`d(A)`
`put_unsafe_value(x(V),A)`	`u(V) d(A)`
`put_unsafe_value(y(_),A)`	`d(A)`
`put_constant(_,A)`	`d(A)`
`put_nil(A)`	`d(A)`
`put_structure(_,A)`	`d(A)`
`put_list(A)`	`d(A)`
`unify_variable(x(V))`	`d(V)`
`unify_value(x(V))`	`u(V)`
`unify_local_value(x(V))`	`u(V)`
`Default`	$\emptyset$

Concept Index

A

abstract syntax tree . . . 8
active environment size . . . 6
active variable count . . . 6, 37
aliasing . . . 32
area, code . . . 47
argument register . . . 50
argument register cache . . . 16
arithmetic expression . . . 24
array, binding . . . 1
assignment, correct . . . 28
assignment, optimal . . . 28

B

basic block . . . 2
binding array . . . 1
binding, conditional . . . 49
binding, illegal . . . 48
binding, unconditional . . . 49
block, basic . . . 2
bound variable . . . 48

C

cache, argument register . . . 16
cell, value . . . 48
choicepoint . . . 49
chunk . . . 3
clause compiler . . . 1
clause, halting . . . 4
clause, iterative . . . 4
clause, recursive . . . 4
code area . . . 47
commit . . . 7
compiler, clause . . . 1
compound term . . . 48
conditional binding . . . 49
conditional graph expression . . . 12
constant . . . 48
copying, structure . . . 48
correct assignment . . . 28
cut . . . 7

D

deep instruction . . . 5
dereferencing . . . 48
descriptor, variable . . . 9

E

environment . . . 49
environment size, active . . . 6
environment trimming . . . 50
environment, shadow . . . 6
expression, arithmetic . . . 24

F

flattening . . . 16

G

get instruction . . . 50, 57
get instruction, indexed . . . 26
get instruction, special . . . 26
global stack . . . 47
global variable . . . 48
globalising . . . 48
goal, inline . . . 3
graph expression, conditional . . . 12
graph, interference . . . 31

H

halting clause . . . 4

I

illegal binding . . . 48
indexed get instruction . . . 26
indexing information . . . 1
indexing instruction . . . 50, 59
information, indexing . . . 1
information, static . . . 1
inline goal . . . 3
instruction set . . . 56
instruction, deep . . . 5
instruction, get . . . 50, 57
instruction, indexing . . . 50, 59
instruction, procedural . . . 50, 59
instruction, put . . . 50, 56
instruction, unify . . . 50, 58

instruction, utility 60
interference 31
interference graph 31
internal predicate 10
iterative clause 4

L

list 48
live variable 28
live variable set 28
local stack 47
local variable 48

M

mode, read 52
mode, write 52
model, SRI 1

N

name, object code 4
number, variable 6, 37

O

object code name 4
operator, pruning 7
operator, use-def 28
optimal assignment 28

P

permanent variable 50
predicate, internal 10
procedural instruction 50, 59
property, single assignment 13
pruning operator 7
pseudo-goal 3
put instruction 50, 56

R

read mode 52
recursive clause 4
reference 48
register, argument 50

S

set, live variable 28
shadow environment 6
single assignment property 13
special get instruction 26
SRI model 1
stack, global 47
stack, local 47
stack, trail 47
static information 1
structure 48
structure copying 48
syntax tree, abstract 8

T

temporary 13
temporary variable 50
term, compound 48
trail stack 47
trimming, environment 50

U

unbound variable 48
unconditional binding 49
unify instruction 50, 58
use-def operator 28
utility instruction 60

V

value cell 48
variable count, active 6, 37
variable descriptor 9
variable number 6, 37
variable, bound 48
variable, global 48
variable, live 28
variable, local 48
variable, permanent 50
variable, temporary 50
variable, unbound 48

W

WAM 1
write mode 52

Table of Contents

1 Introduction 1

2 Preliminaries 3

2.1 Terminology 3
2.2 Instruction Set Extensions 4
2.2.1 Support for Full Prolog 4
2.2.2 Indexing Support 5
2.2.3 Shallow Backtracking Support 5
2.2.4 Memory Management Support 6
2.2.5 SRI Model Support 6

3 Compilation Phases 8

3.1 Generating the Abstract Syntax Tree 8
3.2 Spawning Internal Predicates 10
3.3 Allocating Permanent Variables 13
3.4 Code Generation Proper 13
3.4.1 Overview 14
3.4.2 Flattening Terms 16
3.4.3 Head Arguments 18
3.4.4 Body Goal Arguments 19
3.4.5 Compound Terms 21
3.4.6 Inline Goals 22
3.5 Extracting Indexing Information 25
3.6 Register Allocation 28
3.6.1 Lifetime Analysis 28
3.6.2 Correct and Optimal Assignments 31
3.6.3 Implementation 32
3.6.4 The First Chunk 33
3.6.5 Subsequent Chunks 35
3.6.6 Single Chunk 35
3.6.7 Complexity Analysis 35
3.7 Final Editing 36

4 Optimisations 39

5 Comparison with Other Work 40

5.1 Other Prolog Compilers 40
5.2 Graph Colouring 43

6 Conclusions **44**
Acknowledgements 44

References **45**

Appendix I: Synopsis of the WAM **47**
Data Areas 47
Abstract Machine Registers 47
The Global Stack 48
The Local Stack 49
The Trail Stack 49
Treatment of Variables 50
The Instruction Set 50
Get Instructions 51
Put Instructions 52
Unify Instructions 52
Procedural Instructions 54

Appendix II: Instruction Set **56**
Put Instructions 56
Get Instructions 57
Unify Instructions 58
Procedural Instructions 59
Indexing Instructions 59
Utility Instructions 60

Appendix III: Instructions and Use-Def Operators **62**

Concept Index **63**